CRANFIELD UNIVERSITY

M. C. GIA

DESIGN OF DATA STRUCTURES FOR TERRAIN REFERENCE NAVIGATION

COLLEGE OF AERONAUTICS

DEPARTMENT OF AVIONICS

PhD THESIS

CRANFIELD UNIVERSITY

CRANFIELD UNIVERSITY

COLLEGE OF AERONAUTICS

DEPARTMENT OF AVIONICS

Ph.D. THESIS

Academic Year 1993-4

M C GIA

DESIGN OF DATA STRUCTURES FOR TERRAIN REFERENCE NAVIGATION

Supervisor: Professor D J ALLERTON

May 1994

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

To my parents

ACKNOWLEDGMENTS

I am greatly indebted to my supervisor, Professor D.J. Allerton, for the guidance, suggestions, encouragement and patience shown to me throughout the work.

I greatly indebted to my parents, for their immense love and sacrifices for their children. I would like to express my deep gratitude to my sister You-wen and brother Min-su, for their consistent encouragement and support, especially for looking after our parents when I am studying abroad.

I also like to thank the UK Ordnance Survey for providing the DTM files.

The sponsorship of my study in Cranfield University by the Materiel Test and Evaluation Service in the Republic of China, is gratefully acknowledged.

Last but not least, I would like to express my gratitude to my loving wife, Show-shun, and my son, Chang-lin, for their kindness support, patience, understanding and most importantly, love throughout the years of my graduate study in the United Kingdom.

ABSTRACT

This thesis describes the design of a data structure for use with Digitised Terrain Elevation Data (DTED) in Terrain Reference Navigation (TRN) systems. The data structure is based on a variant of quad-tree and oct-tree data structures to provide an efficient representation of terrain in terms of storage requirements and access operations. These data structure are applied to flight path planning operations in mission management applications. The algorithms developed for flight path planning have been implemented in the C programming language for a standard PC.

Current research in TRN systems is reviewed and attention is given to the use of hierarchical data structures to cope with the potentially large data base needed for DTED files. Data structure combining quad-trees and oct-trees are developed with an emphasis on data reduction using pointerless trees and the use of locational codes to provide straightforward mapping between quad-trees and oct-trees, in other words, between two-dimensional co-ordinates and three-dimensional co-ordinates. Analysis of these algorithms is described for two DTED files to illustrate storage improvements and to verify a set of database access operations.

These data structures are applied to problems of flight path planning where the navigation space comprises objects above a specific altitude and this three-dimensional space is searched for a flight path which avoids the obstacles and satisfies specific operational criteria. Algorithms are developed to extract a visibility graph from the terrain database and to determine the preferred flight path from a set of paths which satisfy defined constraints. Several search techniques are developed which exploit the efficiency of the quad-tree and oct-tree data structures. These methods are extended to real-time flight-path planning where predicted times for access operations are used to direct flight path extraction by varying the tree resolution during computation of the flight path.

A comprehensive set of results are provided to illustrate:

- the storage efficiency of quad-tree and oct-tree data structures
- the application of pyramid structures to represent navigation space
- analysis of the time to compute the visibility graph and to extract flight paths
- integration of these methods with a real-time mission management simulation on a PC

The thesis draws conclusions on the efficiency of these techniques for the representation of DTEDs and to access objects in TRN systems. It is observed that the use of hierarchical data structures in the form of quad-trees and oct-trees offers significant improvement in accessing DTEDS, for future use in TRN systems. The thesis concludes by outlining areas of further work where the techniques can be further developed for applications in mission management and navigation using DTED files.

CONTENTS

FIGURES *page no.*

TABLES

NOTATION

		page no.
M(L),M(L-1)...M(0)	A sequence of array in a pyramid	22
B, W	Black and white nodes of quad-trees	26
m, n	Size of a block, array	26
X	don't care value, merged node	30
NW, NE, SW, SE	Quadrants of quad-trees structure	50
NWB, NEB, SWT, SET	Octants of oct-trees structure	50
(I,J)	The position of an elevation in 2-D space	52
(I,J,K)	The position of an elevation in 3-D space	52
I, J, K, S	Four digit-elements of a node of a terrain oct-tree	59
c_l, d_l, e_l, e_l	The digit in binary representation of I,J,K,S	59
Q, Q_{2d}, Q_{3d}	2-D and 3-D locational code	63
N_L	Number of leaf nodes	71
H_{abs}	Absolute altitude	85
H_{true}	True altitude or flight altitude	85
S, G	Start and goal points	90
SEED	Points along a direct path collide with danger region	90
N_{start},N_{goal}	Start and goal nodes	97
N	a node, search key, locational code	97
$LIST_{node}$	List of nodes	97
N_{dn}	Number of danger nodes	100
N_{lp}	Number of point elements along a direct path	100
p	Perimeter of a region	101
W, W_{from}, W_{to}	Waypoint, Start and end points of a path segment	101
VG	Visibility graph	101
W_{points}	Waypoint	104
A^*	A^* algorithm	107
M_d	Danger nodes list	134
N_{node}	Number of nodes	136
T_{wp}	Time to locate the waypoints	138
T_{vg}	Time to construct the visibility graph	138
T_{sp}	Time to search a path	138
T_{total}	Total time to find a path	138
$P_{segment}$	Number of path segment in a visibility graph	138

CHAPTER 1

INTRODUCTION

1.1 Aircraft Navigation

In airborne navigation, position and flight state information is required at all times with a high degree of accuracy. For that purpose, many systems of determining navigation information have been devised. The position-determination schemes can be classified as either *dead reckoning* or *position fixing* [Kayt69]. Dead reckoning consists of extrapolation of a 'known' position to some future time. It involves measurement of direction of motion and distance travelled. In contrast to dead reckoning, position fixing is the determination of the position of the craft (termed a *fix)* without reference to any former position.

Dead reckoning can be considered as the basis of aircraft navigation, with position fixing constituting a method of updating navigation data [Kayt69]. An aircraft navigation system utilizes both dead-reckoning and direct-position data. Actually, dead reckoning and position fixing complement each other, each providing an independent means of checking the accuracy of the other. The navigation computer combines the direct-position fixes and dead-reckoning data with an estimate of the aircraft's position and velocity. The following give some examples of modern navigation systems :

1. Very High Frequency Omni Range (VOR) is a ground-based radio system which provides the pilot with an accurate determination of bearing to a ground station. Because VOR operates in the very high frequency bands (108-117.95 MHz) it is not subject to atmospheric disturbances, but it is subject to 'line-of-sight' constraints, and has a usable range between 30 and 200 miles depending on aircraft altitude [Cann76]. VOR range increases with aircraft altitude but the accuracy of a VOR also deteriorates with distance.

2. Distance-Measuring Equipment (DME) accurately measures slant range from an airborne transmitter/receiver to a ground station. DME is now an integral part of accurate navigational guidance, whether linked to a VOR facility to provide range and bearing, or scanning a number of DME stations to determine aircraft position.

3. The Global Positioning System (GPS) is a network of 24 satellites based on measuring a timing pulse from satellites transmissions and is a highly accurate ranging system. Twenty-one of the satellites are operational, with three standbys, and the network will provide coverage over most of the earth's surface. The system will provide a world-wide navigation aid with an accuracy better than 100 metres and is the most accurate positioning fixing system developed to date [Payl93].

4. An Inertial Navigation System (INS) determines position on a continuous basis with additional data being needed to re-align the system periodically. INS determines the displacement from the point of departure by measuring the accelerations exerted upon a gyroscopically stabilised platform as a result of aircraft motion. Once an INS is supplied with initial position information, it is capable of continuously updating displays of aircraft position, ground speed, attitude and heading.

An INS exhibits significant drift with respect to time. This disadvantage is due to errors in the initialization of the inertial measurement unit and inertial sensor imperfections such as accelerometer bias and gyroscope friction. This drawback can be overcome by re-aligning the position along the flight path from visual or radio fixes [Siou93].

Most modern aircraft rely on an INS as a means of estimating an aircraft's position and heading. With the advent of faster and more advanced aircraft, navigation has required increased accuracy and INS has been used in aircraft for long-range flights

to provide a global navigation capability for both commercial and military applications. Inertial Navigation Systems have undergone a remarkable revolution, progressing from simple dead-reckoning to modern techniques which integrate pure inertial systems with additional sensors, particularly GPS.

Various methods for determining fixes, including radio navigation methods (Radio compass, Omega, Loran, Decca, VOR, DME, TACAN) which determine the actual position by reference to fixed ground stations, have been integrated with Initial Navigation Systems to take advantage of size, weight, and power saving. The current generation of INS afford high quality and performance, with drift rates typically better than 1km/h [Siou93].

The tools used for navigation involve the use of radio transmissions and are, therefore, range-restricted. In the case of systems that operate in the higher frequency bands, they are also constrained by the 'line-of-sight' rule. That is, that the characteristics of radio waves normally travel in a straight line unless they are 'bent' by atmospheric phenomena, reflected by dense objects, or screened by high terrain. In most cases, the range of navigation aid used increases with aircraft altitude.

1.2 Terrain Referenced Navigation

Recently, some avionic systems have employed a terrain elevation database for navigation or guidance. These techniques are termed Terrain Referenced Navigation (TRN). TRN is based on accurate knowledge of aircraft position, currently provided by GPS and inertial navigation systems, combined with accurate knowledge of the terrain topology which is derived from digitised terrain maps.

TRN is not a stand alone system; it is combined with other navigation systems [Henl88, Prie90] and operates by measuring the terrain height below the aircraft and comparing this measurement with heights derived from a digital terrain database. The comparison is performed in a correlation process which provides accurate position and

height corrections for a dead reckoning navigation system.

The accuracy of the TRN system depends on map accuracy and terrain variation, but the system has characteristics which make it complementary to other methods [Camb85]. For example, TRN can be used when the full accuracy of radio aids (e.g. GPS) can not be maintained due to aircraft position, for example in hilly or mountainous terrain. Since TRN does not depend on external aids, the gravity error does not exist in a TRN system, and it is difficult to detect and jam.

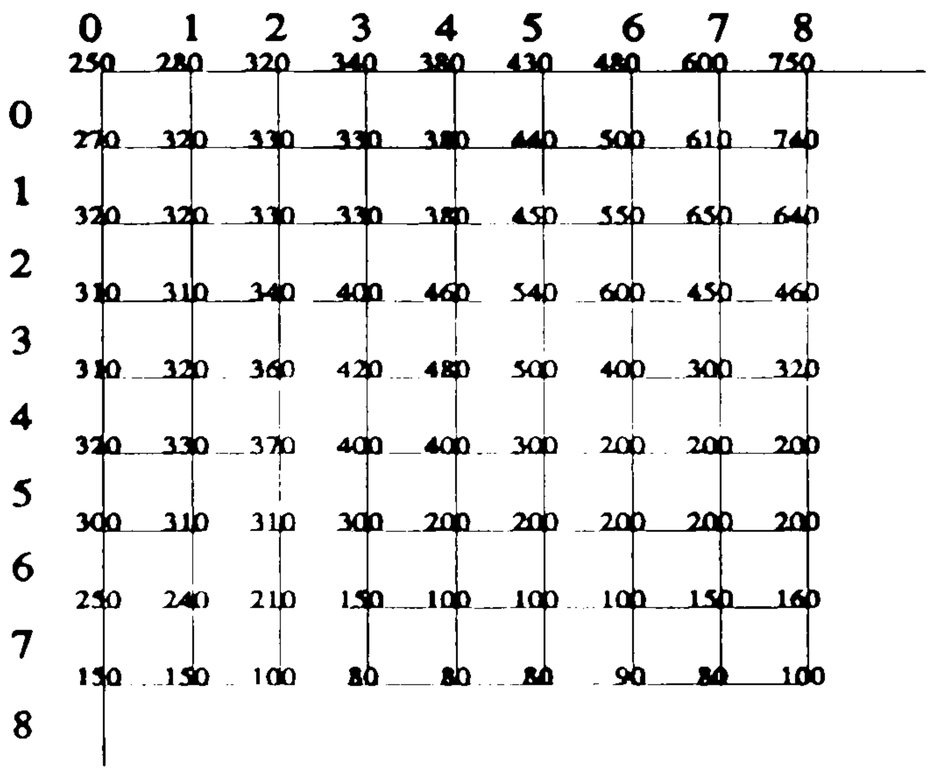
For instance, the Sandia Inertial Terrain-Aided Navigation (SITAN) [Camb85] utilizes radar altimeter returns, a terrain elevation database, and a control filter to calculate corrections to the aircraft's inertial navigation system. SITAN reduces the INS position error from the order of several miles per hour down to the order of 100 metres or less. This accuracy improvement comes through making periodic adjustments to the INS computations by comparing ground clearance measurements with a predicted ground clearance, as determined from the Digital Terrain Elevation Data (DTED).

The increasingly dangerous air defense environments dictate a requirement for low level and high speed aircraft navigation while minimising exposure to threats, therefore, many DTED based applications are integrated with aircraft navigation systems. For example, a DTED can provide elevation data for trajectory and guidance applications ranging from simple straight-line Terrain Following (TF) between waypoints to sophisticated 'ground-hugging' meandering flight [Barn84, Prie90, Zele92, Barf93].

Other applications of stored digital terrain data with reference to navigation, include terrain ground collision avoidance, obstacle cuing, threat coverage zones, passive ranging, ground proximity warning (GPWS), generation of terrain perspective for 3-D viewing, radar profile predictions and refined digital map displays [Webe84, Benn88, Prie90].

1.3 Terrain Elevation Data

Digital Terrain elevation data is used in aircraft navigation for real time navigation to increase the confidence in parameters such as aircraft position and velocity



(a)

(b)

Figure 1.1 (a) A sampled terrain elevation data and its (b) DTED file arrangement.

[Barn 84, Burn 84, Camb 85]. Usually, DTEDs are stored in memory modules in the form of blocks of elevation values representing each point of a matrix of plan positions as shown in Figure 1.1. Typically, the DTED resolution is of the order 100 metre or 3 arc seconds intervals [Waru84].

As an aircraft proceeds along its track these blocks are read into fast local memory to form a matrix of terrain elevation values for an area surrounding the aircraft [Cree86]. This matrix can be used to determine the height of the terrain at any point of an array of offset points around the current position of the aircraft.

Digitised terrain data occupies large amounts of memory. For example, if a terrain is mapped at 100 metres intervals over a region 10000 Km by 10000 Km, there are 10^{10} grid points if the terrain is represented by a two-dimensional array defining elevation at regular grid points. Clearly, the computations inherent in continuously

accessing 10^{10} grid points are formidable and it is necessary to reduce, or compress, this information.

There is natural redundancy in terrain data, that is to say, large regions of terrain may exhibit some common features; for example, a large flat area of terrain below fifty feet could be represented as a single object. More significantly, data represented as a regular array lacks topological structure. For example, it is not possible to extract a region of terrain above a specific altitude, without accessing the whole array. In many applications, it may be beneficial to represent terrain at a coarser resolution than that given by the grid points and this is clearly difficult to achieve from data stored as a two-dimensional grid.

Digital terrain elevation data is required in terrain referenced navigation systems. But clearly, the amount of bulk on-line storage required for the DTED can become a limiting factor in system design. Usually, on-line storage of a navigation system cannot contain the entire DTED for a gaming area in a single memory. In order to achieve the efficiency and speed requirements in a real-time airborne environment, a DTED is partitioned into sub-blocks, then individually compressed and stored on disk [Camb85]. During navigation, the sub-blocks are reconstructed as needed, in real-time.

By means of data compression techniques, the problems of a grid approach, in terms of the large storage requirement, are partially relieved. However, the inability to exploit the redundancy, the lack of flexibility and the deficiency of providing global representations of terrain at coarser levels still remains.

For long distance flights, complete surface references are needed with acceptable accuracy for the entire area of navigation. Compact and accurate terrain data representation is a basic requirement of TRN applications. Trees are widely used to represent data structures which exhibit hierarchical characteristics. Nodes within a tree can 'summarise' information which is contained in lower nodes of the tree. This representation of information can exploit the physical structure of specific data and thus

reduce the number of access operations to retrieve data from the tree.

A hierarchical representation can be applied to modelling the terrain with sufficient accuracy to perform such tasks as obstacle avoidance, path planning, and long-range mission planning and with sufficient speed to operate in real-time. The design of a compact terrain representation which is based on hierarchical data structures, and which in turn affords the ability to efficiently manipulate terrain data, is the primary aim of this thesis.

1.4 Terrain Elevation Data Structures

Improvements in both hardware and software technologies have contributed to advanced airborne capability. The application of digital technology has had an enormous impact on the performance and capabilities of avionic systems. The increase in component densities (a factor of one hundred thousand in two decades [Midd89]) which has been achieved has not only reduced the space required for airborne equipment but also increased the speed of operation. With the advent of powerful and embedded real-time computing systems during the past decade, airborne computers have become smaller, faster, cheaper and more reliable.

However, software developments have not kept pace with hardware developments and deficient software leads to deficiencies in performance. Together with advanced computer technology, standardized programming languages and software, development methodologies have also been developed to support advanced navigation systems. As part of this software technology, data structures play a key role in any applications because they dictate how the data is collected for storage, how it is stored, accessed and retrieved, and how it can be manipulated for problem-solving.

The main objective of designing an airborne terrain data structure is to define a representation which minimizes the effort required to store, access, display and process the data base for a real-time airborne environment. The data structure must

meet the requirements for the high speed of response necessitated in continuous navigation updating. In this thesis, the design of an appropriate data structure to represent terrain elevation data concentrates on access protocols, space requirements, regularity of manipulation, associated operations and algorithms.

The Geographic Information System (GIS) is a system for the measurement and analysis of the Earth's surface. An important aspect of this application is the way in which spatial data (which in GIS consists of points, lines, rectangles, regions, surfaces, and volumes) is organized. Terrain elevation is the primary spatial data contained in GIS and considerable attention has been given to terrain data structuring methods [Mark84, Same84a, Cebr85, Meno87, Shaf89]. The techniques that are used in GIS to organize large matrices of elevation data provides the background for airborne terrain data structure design developed in this thesis.

Since digital elevation data is stored in a two-dimensional array, it is analogous to an array of image pixels with coefficients defining terrain features attached to the corresponding pixel position and possibly, techniques which are used in processing image data can also be applied to terrain elevation data.

1.5 Representation of Terrain Data in Navigational Space

The primary function of an aircraft mission management system is to determine a flight path which satisfies a set of operational constraints and to keep the pilot informed of the progress of the flight plan. The pilot is provided with information to determine distance (and time) to way points, guidance cues, fuel management support and a capability to monitor and to organise the flight plan.

Although it is possible to perform flight planning computations on a ground-based workstation and to up-load mission data prior to takeoff, in practice, mission capability is significantly improved by the provision of on-board mission management, subject to the limitation of the performance of the mission management computer to

solve the real-time computations inherent in the navigation calculations.

Once an aircraft flight path and the topology of a region of terrain can be readily determined, advanced forms of aircraft navigation become feasible including terrain avoidance, terrain following, threat avoidance and terrain matching. The common theme with all these applications is that navigation entails computation of the aircraft flight path with reference to the topology (or three-dimensional geometry) of the terrain and this implies access to a database holding the terrain data.

One of the basic tasks required in aircraft navigation is to fly from one location to another while avoiding all the obstacles along the path. For a grid file with even spacing but arbitrary distributed elevation values, each data element represents point information. Initially, each 'grid post' is discrete and no adjacency relationships exist to define a polygon obstacle area with common topographic (elevation) properties. However, the co-ordinates of each grid point are 'implicit' in the grid ordering and the grid is 'area-covering' and directly indicates a point's immediate neighbours.

In the grid file representation, a terrain can be depicted as a composite of three-dimensional polygons which have a one-to-one correspondence with the grid elements. A polygon with its elevation over a given minimum flight altitude is defined as an obstacle area to an aircraft. This process can readily be visualized by considering a projection of the three-dimensional polygons to the two-dimensional polygons, where each polygon is either a free space or an obstacle to an aircraft as shown in figure 1.2.

In the two dimensional plane there are only three basic classes of objects: points or nodes, arcs and polygons. Thus a polygon may be defined by several boundaries, by several nodes, (which occur at the junctions between boundaries) and also by the adjacent polygons that bound it. Line segments or arcs may be defined in terms of the two end points (nodes). When these basic classes of objects are used to represent navigation space, the topographic information can be easily depicted. For example, a polygon danger area is represented by a group of connected 'area-covering' grid points

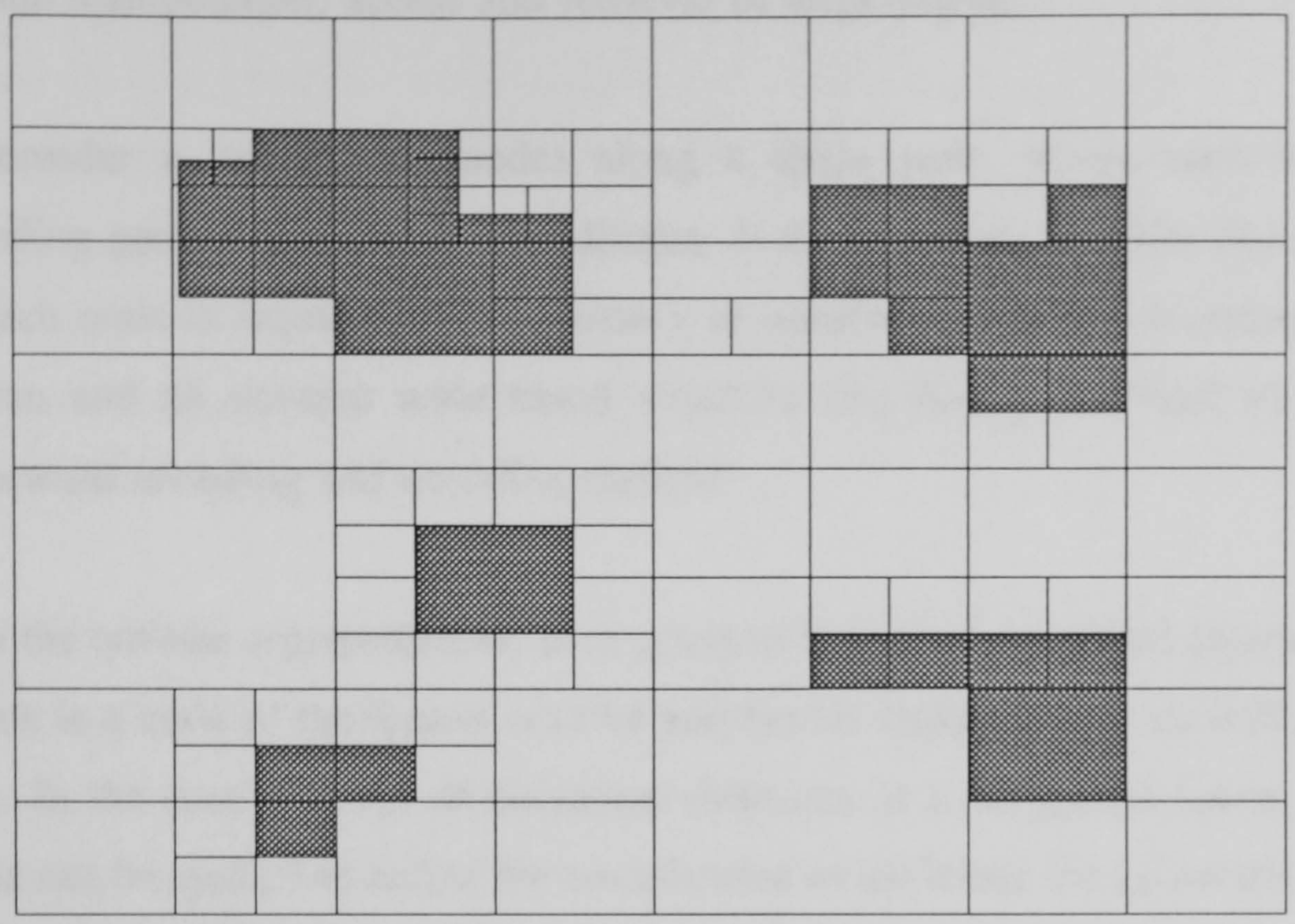


Figure 1.2 A two-dimensional representation of navigation space, the shaded area represents obstacles, unshaded area as free space.

with equal elevation data; a flight path segment between two points is represented by a set of grid points.

A polygon set is analogous to a graph. A graph is formed from regions, edges and nodes, which are directly related to the polygons, arcs and nodes but do not have implicit coordinates. It is proposed in this thesis that the navigation space is mapped from a co-ordinate based grid file to a geometric graph representation of navigation space. Once a graph structure is available, many path planning operations are performed on the graph to gain improvements in storage and speed.

In this thesis, a data structure termed a 'terrain oct-tree' is devised to reorganize the terrain elevation data. As a terrain is strictly a single-valued continuous surface, grid points with equal or approximate elevation can be aggregated to a single data item (or block) according to their planar position. The terrain surface is thus a composition of blocks. It is proposed that the use of an oct-tree terrain data representation may

improve the manipulation, access and retrieval of terrain data.

Consider a set of tree nodes along a flight path, where each node has corresponding geometric *X* and *Y* co-ordinates. If the geometric position and elevation data of each node is required, it is necessary to transform between co-ordinate based information and an oct-tree node based structure and this is achieved by using a straightforward encoding and decoding method.

In the oct-tree representation, each polygon is a set of connected square blocks. Each block is a node of the terrain oct-tree and has its representative co-ordinates and elevation. In the case of a set of polygonal obstacles in a navigation space, a set of operations can be applied to define the co-ordinates which locate the geometric position of the obstacles. The set of operations is also considered as a translation process between co-ordinate based information and an oct-tree node based structure. The development of oct-tree representation, the transformation and implementation processes are covered in the following chapters.

1.6 Objectives of the Research Programme

As digital terrain elevation data has been used extensively in airborne navigation, this research is intended to design an efficient terrain data structure for real time terrain reference navigation which overcomes the disadvantages of DTED in both storage requirement and operational performance.

This project investigates algorithms to represent and to process digitised terrain databases using oct-trees to provide a compact representation of the terrain data and to allow data to be processed at varying levels of resolution, so that the hierarchical decomposition of the terrain can reduce the overall number of geometric computations in navigation applications.

The fundamental problem of flight path planning is to extract a set of potential flight paths to determine an optimal flight path which satisfies specific constraints, which may include aircraft manoeuvrability, aircraft performance, air-traffic regulations and operational requirements. In extracting flight paths from a terrain database, it is essential that the terrain data is organised efficiently to minimise delays in accessing terrain elevation data. The TRN based flight path planning problem is investigated by using the following strategies:

1. Designing an efficient terrain representation which is used to reorganize the large amount of DTED and is able to represent the real-time airborne navigation space efficiently in a global manner without losing the accuracy of the original DTED file.

2. Designing a real time dynamic approach in order to achieve the requirements of the flight path plan generation in which the terrain elevation data needs to be referenced and retrieved frequently from the data base according to flight conditions.

The flight path planning approach discussed in this thesis has the following features:

- Navigation space is only defined in an encoded terrain oct-tree representation, no other format of terrain topographical information is provided.
- The obstacles are accessed by the flight path planning algorithm in real time during the mission.
- The flight path is based on a collision check approach which guarantees that the path generated is collision free.

In order to demonstrate the operational efficiency of the application of terrain oct-tree structures, this thesis also discusses the merits of the oct-tree approach with reference to the performance analysis of a real-time flight path planning system. The study outlines the future potential of the terrain oct-tree representation.

The following discussion in the thesis includes a detailed description of the theory of terrain oct-tree design along with experimental and analytical results. The results support the feasibility of the terrain oct-tree model and its applications which can be integrated in a TRN system to achieve storage efficiency and speed performance.

1.7 Summary of the Research Contributions

In this study, the following achievements are reports:

1. A novel terrain representation based on hierarchical data structure is developed with the potential to reduce the time to access objects in the terrain database. The structure also affords data compression and simplifies the operations on the data structure. Moreover, basic geometric operations such as neighbour locating, distance measuring and region expansion can be directly performed on the compressed data without reconstruction of a DTED.

2. The terrain oct-tree and terrain pyramid approach to terrain representation presented in this thesis simplifies the design of many aspects of DTED based navigation systems. For example, the terrain oct-tree is treated as a collection of leaf nodes and is stored in the form of sorted lists to facilitate list processing operations including insertion, deletion and searching. Furthermore, as a node in a terrain oct-tree contains topological data, a navigation display can be realised by combining the data structure of terrain oct-tree with the data structure of a graph representation.

3. The terrain oct-tree and its planar projection codes allow three-dimensional path planning to be performed in two-dimensional space. The projection codes provide an index to access the DTED file if the original accuracy of terrain data is required. Many other three-dimensional applications such as line-of-sight determination, danger area masking can be achieved in two-dimensional space.

4. Both the variable resolution and multiple resolution characteristics of hierarchical data structure are used to improve the storage requirement and time performance in the flight path planning algorithm. The results show that the use of variable and multiple resolutions has led to a simplified navigation space representation without loss of accuracy. The reduction of nodes in a terrain oct-tree in turn reduces the computation costs associated with the path planning process.

5. In terms of motion planning problems, the proposed terrain oct-tree is categorised as a cell decomposition approach. Instead of using the connectivity graph method adopted by most cell decomposition terrain models for path searching, the partial visibility graph approach has been adopted for the search space representation to reduce the real-time computational requirements.

1.8 Thesis Organisation

This thesis is organized as follows. Chapter 2 introduces some of the basic aspects of hierarchical data structures. As terrain data is generally stored in a matrix format similar to image data, the principles of hierarchical data structure operations such as regular decompositions, quad-trees, oct-trees and pyramids used in image processing are reviewed. An alternative implementation of quad-trees is developed which does not use pointers. The key characteristics of quad-tree (oct-tree) structures are considered for use in terrain representations. Basic path planning problems are introduced and the literature on path planning problems is reviewed together with the modelling of terrain for the obstacle avoidance problem.

Chapter 3 describes the features and disadvantages of traditional terrain matrices. The requirements of terrain oct-tree design are addressed. The chapter contains details oct-tree data structures and both construction and manipulation of a terrain oct-tree. The algorithms are described in pseudo-code and their storage requirements and time performance are evaluated.

Chapter 4 begins with examination of the navigation space and the requirements of a flight path; the difference between robot motion planning and flight path planning are discussed. A navigation space is provided in a terrain oct-tree representation for the flight path planning which is constrained by minimum flight altitude and shortest distance. In particular, the hierarchical features of quad-tree and oct-tree are applied to improve the time performance during the planning, where the path searching space is reduced by using a partial visibility graph.

As it is anticipated that an aircraft flight path needs to be modified in real-time to match flight conditions and changing obstacles in the environment, the strategy of applying the proposed flight path planning algorithm to a real-time dynamic environment is discussed in chapter 5. The off-line results of the computation times obtained in chapter 4 are subsequently used as references for adapting the real time dynamic flight path planning algorithm to a given terrain. A hierarchical data structure is introduced to represent multiple-resolutions of a terrain oct-tree.

Chapter 6 presents the experimental results of the algorithms described in chapter 3, 4 and 5. The storage requirements and the data reduction rate between a DTED file and a DTED encoded terrain oct-tree are compared. The results for the terrain oct-tree encoding include different scaling factors and resolutions.

In the application of terrain oct-trees to flight path planning problems, the interim results and the observations of the off-line computation performance are discussed. The graphical display of encoded terrain oct-trees and the simulation results for the real-time flight path planning are presented.

Chapter 7 gives a brief summary and conclusions of the thesis and suggests potential areas of future research.

CHAPTER 2

LITERATURE REVIEW

2.1 Properties of a Terrain Matrix

Terrain Referenced Navigation systems incorporate Digital Terrain Elevation Data to provide accurate position and height corrections for a dead reckoning navigation system [Henl88, Prie90]. The matrix or two-dimensional array of elevation values is stored in a digital form to represent terrain data and is provided in common format [Boys86].

DTED typically consists of elevation values, relative to mean sea level, arranged in rectangular grids or "cells" in latitude and longitude. For instance, Figure 2.1 depicts a diagram of the grid positions along with their elevations as the spatial peak points in three-dimension space; the point values represent a small sub-block of a DTED.

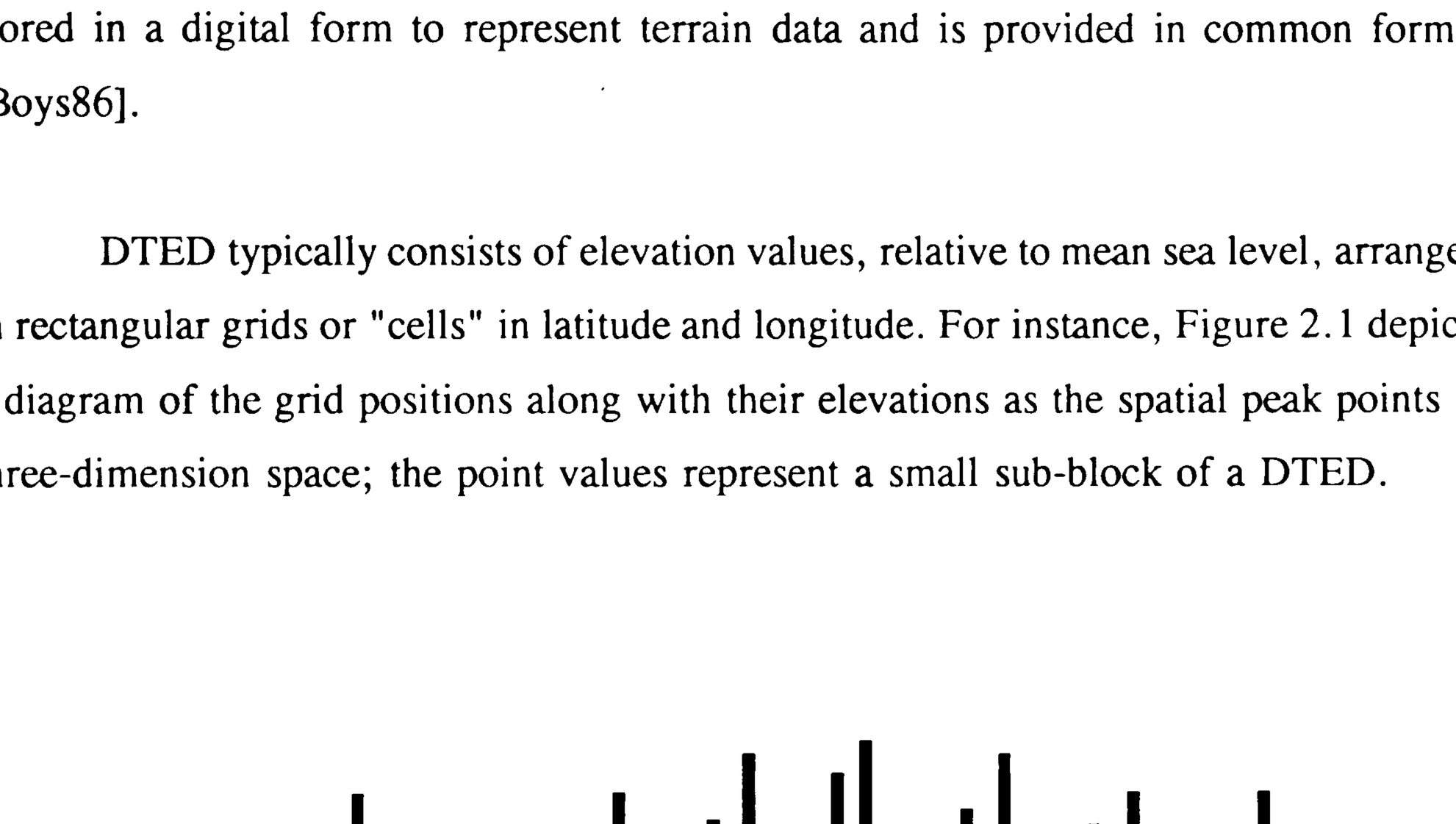


Figure 2.1 An example of scaled terrain grid file diagram in 3-D.

For example, the DTED used in this study is the OSGB 1:50000 Scale Height Data termed Digital Terrain Model Data (DTM) obtained from Ordnance Survey. The DTM file provided by the Ordnance Survey consists of height values at each intersection of a 50 metre horizontal grid where the values have been mathematically interpolated from the contours on the OS Landranger maps. Heights are ordered reading south-north from the south-west corner of the 30 Km square at 50 m intervals, with 601 points to a column. This column is then followed by 600 subsequent columns, giving 361201 height values for a 30 km square DTM tile. One set of DTM has been provided for an area of the Peak District, the other set of DTED is for an area of Port Talbot.

Accessing terrain elevation data is straightforward using arrays. Since an elevation array consists of only the altitude of the surface at each sample point, geographic locations are determined by grid spacing and are implicit in the sequential positions of the altitude values within the storage array. From the digital terrain matrix, the altitude of a particular point (x,y) of the terrain can be directly accessed by simply making a row and column index pair from x and y, and retrieving the array coefficient indicated by the pair. A further advantage of the matrix form of terrain representation is that the neighbours of a given point are easily located due to the explicit location information.

The principal disadvantages of a grid approach is the amount of storage required to represent the terrain and the inability to exploit the redundancy. A terrain matrix may contain many elements with equal altitude covering a large area of flat terrain, while the grid must be sufficiently detailed to portray the smallest terrain feature of interest. Terrain data stored in this form usually occupies large amounts of memory, and the amount of bulk on-line storage can become a limiting factor in the design of real-time navigation systems.

The lack of flexibility is another principal disadvantage of a grid representation, for example when the improvement of resolution requires the grid spacing to be decreased or points per unit area to be increased. Another problem arises when

displaying an array data base on which the grid file can not give multi-resolution representations of terrain. The multi-resolution representation is the ability to depict the terrain in various detail levels according to requirements. For instance, to achieve the efficiency and speed requirements in an airborne environment, it is not necessary to display the finest level of a terrain representation, a representation with reduced-resolution of terrain may be adequate.

To achieve the efficiency and speed requirements in an real time airborne environment, data compression algorithms are applied to improve the utilization of DTED and partially relieve the redundancy problem. For instance, the Sandia Inertial Terrain-Aided Navigation (SITAN) systems apply an image data compression technique termed two-dimensional discrete cosine transformation (DCT) combined with linear scaling, to pack the original data into the lower frequency transform coefficients [Cree86]. Data compression is achieved by discarding transform coefficients of negligible amplitude. This is done by using a variance criterion as a low-pass filtering process to judge the relative significance of a particular transform coefficient. The compression ratio depends on the number of transform coefficients retained out of the original number of elements in a given DTED.

A DTED is partitioned into 16 x 16 or 32 x 32 sub-blocks and individually compressed before storage to effect real-time reconstruction. During navigation, the sub-blocks are reconstructed and 'pieced together' as needed [Camb85, Chan85]. However, the data compression method in SITAN is purely used to reduce the total number of bytes required to store a DTED which covers a given geographical area. This data compression method does not introduce any data structure other than a matrix to provide efficient manipulation on elevation data. The major disadvantages discussed above still remain, particularly in that it does not provide the global representations of terrain at a coarser level.

Many of the deficiencies of the grid file described above demand a better data structure for terrain elevation data. The main objective of airborne terrain data structure

design is to obtain a satisfactory representation, which minimizes the effort required to store, access, display and process the data base for a real-time airborne environment. The objective of the design is thus defined as the design of an effective terrain elevation data structure as well as the design of data compression algorithms which can meet the requirements for a high speed of response and continuous navigation updating.

2.2 Hierarchical Data Structures

2.2.1 Decomposition of Image Data

The image data are represented as matrices with integer elements termed *pixels*. The memory requirement associated with storing these matrices tend to be quite large, for example, 1M bits memory is required to store a black and white image with 1024 x 1024 pixels. The manipulations of digital images require significant amounts of storage and access operations and accordingly, data is not always stored as simple matrices, and more elaborate data structures are frequently used to compress images [Tani75, Hunt79, Sloa79, Klin79, Same90].

Recently, decomposition strategies have been successful in the applications which are most concerned with the large memory requirements associated with storing image data [Tani75, Garg82, Abel84, Lauz85]. It is achieved by implementing a divide-and-conquer approach on the image data where the image is successively divided into smaller homogeneous rectangles. The use of these strategies provides image data structuring methods with hierarchical characteristics. The nodes within this tree-like structure can 'summarise' information which is contained in lower nodes in the tree.

Hierarchical data structures have gained acceptance because of their ability to reduce the space requirements and to focus on the relevant subsets of the data. This compression can result in efficient representation and in faster algorithm execution times as compared to non-hierarchical structures [Same80a, Abel84, Mark85]. The hierarchical data structures discussed in this chapter include regular decompositions,

pyramids, quad-trees and oct-trees.

Before the various decomposition methods are described, some of the conventions and terms with respect to two-dimensional image data are predefined as follows:

1. Image data is assumed to be in the form of a $2^n x 2^n$ (n=0,1,..) array of pixels.
2. Each pixel in the array can be black or white with a corresponding value 1 or 0. The array of pixels is called a *binary image*.
3. For a multi-colour or grey-scale image, the value of each pixel is represented by predefined colour codes.
4. The collection of all spatially adjacent black pixels of the binary image are called *regions* whereas the remainder of the image is called the *background*.
5. The origin of an image is assumed to be (0,0) appearing at its upper left corner; and pixel co-ordinates are referred to by a (column,row) pair.

2.2.2 Regular Decompositions

There are many planar decomposition methods such as triangles, squares, and hexagons [Tani80]. Squares are most used because the resulting decomposition satisfies the following properties:

- It yields a partition that is an infinitely repetitive pattern and therefore it can be used for images of any size. The smallest element in two-dimension space is a pixel. In three-dimension space, the smallest element is a voxel.
- It yields a partition that is infinitely decomposable into increasingly finer resolutions.

The principle of regular decompositions is described as a tree structure by [Klin76] as follows :

An original image array of dimensions $2^n x 2^n$ is assumed. A root node in a tree is constructed that corresponds to this array. The image is then subdivided into either

quadrants or nine parts according to the branching factors along the side of the array, and nodes are set up for each subimage. The tree is built up until the refinement of the image reaches some desired level. If the decomposition does not reach the pixel level, the leaf nodes of the tree are associated with the corresponding size of sub-images.

The principle of regular decompositions has been used in many areas. In computer graphics, hidden-line and hidden-surface elimination algorithms use a recursive decomposition of the picture area [Warnock's algorithm]. The picture area is repeatedly subdivided into rectangles that are successively smaller while searching for areas that are sufficiently simple to be displayed. In image processing applications, when an image is too large to be stored in main memory, the regular decompositions method is often used to store the sub-image blocks in some known sequence on secondary medium, and to access the blocks as they are required for processing [Klin76].

2.2.3 Pyramids

While regular decompositions are performed in a top-down manner, a pyramid is constructed bottom-up, one layer upon another. A pyramid data structure consists of several levels, numbered 0-L, where each level is a two-dimensional raster image. Level L is the most detailed (finest resolution) image; the other levels are derived from lower levels by various approximations [Sloa79].

It is implicit that a pyramid is a multi-resolution representation of an image, where each level in the pyramid represents an image by its approximated array (i.e,. derived from lower level). In other words, a pyramid allows the ability that an image to be processed at different resolution levels. This multi-resolution property has also been applied to problems of feature detection and segmentation [Tani75, Levi80].

The primary advantage of such a scheme is that global structure in the image becomes apparent very early in the display process, allowing a user to begin to examine the image, and interrupt the display when satisfied with the approximation. The

disadvantages are the increased storage and computation costs [Sloa79]. The matrix-sequence definition of pyramids given by [Tani80] is as follows:

A *pyramid* (or pyramid in the matrix-sequence) *P* is a sequence {*M(L),M(L-1),..., M(0)*} of arrays where *M(L)* represents an original image, *M(i-1)* is a version of *M(i)* at half the resolution of *M(i)*, and so on, with *M(0)* a single pixel. The value of a pixel at level *k* of a pyramid is a function of the values of the pixels of a m x n window area in level $k+1$, where m and n is the side length of a window.

The reduction rule can be any function of the pixels in the window (e.g., Minimum, Maximum, Mean, Median, Mode, Sum, Selection) and depends on its applications. For example, if a pyramid structure is applied to the terrain elevation, because elevation data is critical in most aircraft navigation applications, the Maximum function (the highest elevation data in the window area) is the most appropriate function to approximate the terrain surface, allowing the safety altitude can be determined as the elevation data is retrieved.

A pyramid can also be defined in the form of a tree [Tani80]. Assuming a $2^n x 2^n$ binary image is recursively subdivided into quadrants until reaching the individual pixels, the leaf nodes of the resulting tree represent the pixels, while the nodes immediately above the leaf nodes correspond to an image of size 2^{n-1} x 2^{n-1}. The non-leaf nodes are assigned a value that is a function of the lower nodes. This function is the same as the function defined in a matrix-sequence pyramid. Figure 2.2 shows an example of pyramids.

The simplest application of pyramids is to provide reduced-resolution versions of an image. Ideally, the processing time to transform a 64*64 array reduces the processing time for a 128*128 array by a factor of 4. However, the reduction of resolution has an effect on the visual appearance of edges and small objects. In particular, at a coarser level of resolution, edges tend to get smeared, and region separation may disappear [Tani76].

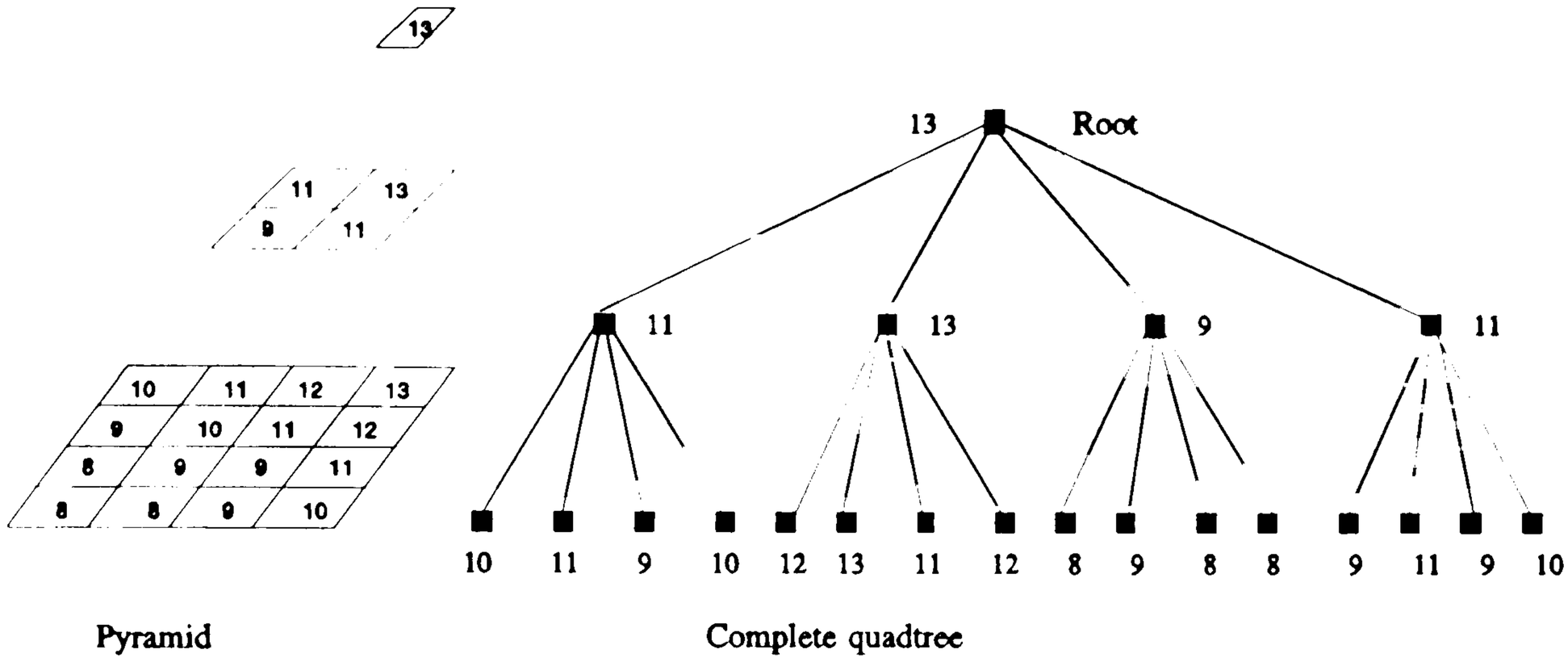


Figure 2.2 Pyramid structure and corresponding complete region quad-tree.

Searching an image is an operation that can be easily performed on pyramids since they can be used to limit the scope of the search. Once an item of information is found at a coarse level, the finer resolution levels can be searched [Tani75]. For example, to locate a white spot in a tree pyramid representation of a binary image, the search begins at the root of the pyramid and at each node, moves to the descendent node with a non-zero value, the search terminates at the pixel level. Moreover, the co-ordinates of the spot can be directly computed from the search path through the tree.

2.2.4 Quad-trees and Oct-trees

The term *quad-tree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space [Klin76, Same90a]. The prime motivation for the development of the quad-tree is the desire to reduce the amount of space necessary to store data by aggregation of homogeneous blocks. An explicit pointer based structure was used to describe quad-trees in most early studies [Hunt78,79, Same90a]. The pointer based

quad-tree, also referred to as the *regular quad-tree*, is described as follows [Same90a]:

The quad-tree is a hierarchical tree representation of a binary image in which each node has either four or zero descendants. The root represents the entire image and the leaf nodes represent maximal blocks of uniform colour. The tree is constructed by repeatedly dividing the image into subquadrants, until blocks of uniform colours are obtained. At this stage, the blocks are represented by leaf nodes along with the colour information. A node of the tree is represented by a record with six fields, four pointers to its four quadrants (NW, NE, SW, SE), a pointer to its parent node and the colour information. Figure 2.3 shows an implementation of the above quad-tree structure.

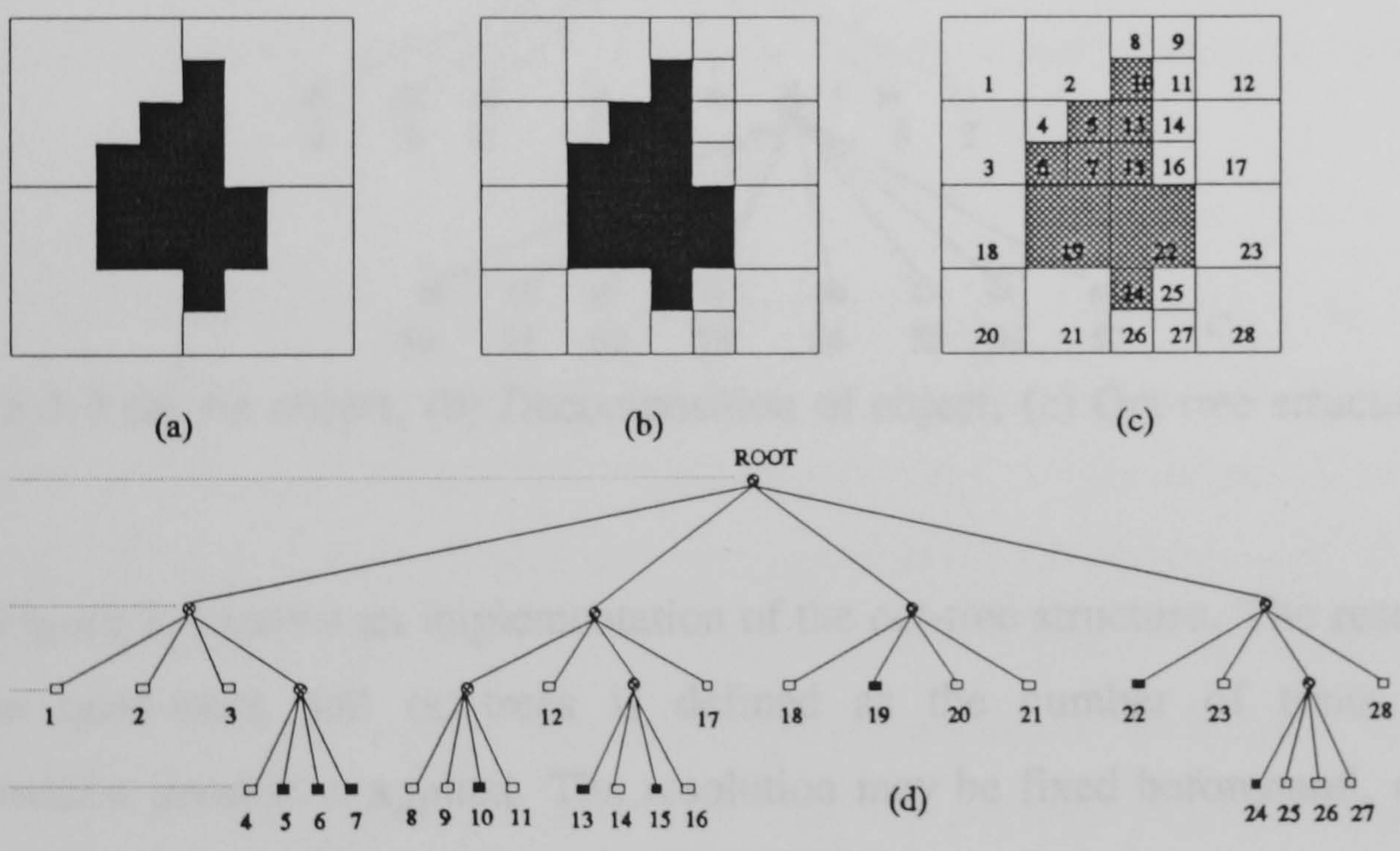


Figure 2.3 (a),(b) Quad-tree decomposition, (c) Quad-tree leaf node numbering, (d) Quad-tree structure.

The quad-tree is easily extended to represent a three-dimensional binary array and the resulting data structure is called an *Oct-tree* [Hunt78]. The oct-tree is defined in an analogous manner to the quad-tree. The oct-tree is constructed by the successive subdivision of a $2^n x 2^n x 2^n$ object array into octants. If the array does not consist entirely

of 1s or 0s, it is subdivided into octants, suboctants, and so on, until cubes (possibly single voxels) are obtained that consist of 1s or 0s. This subdivision process is represented by a tree of degree 8 in which the root node represents the entire object and the leaf nodes correspond to those cubes of the array for which no further subdivision is necessary.

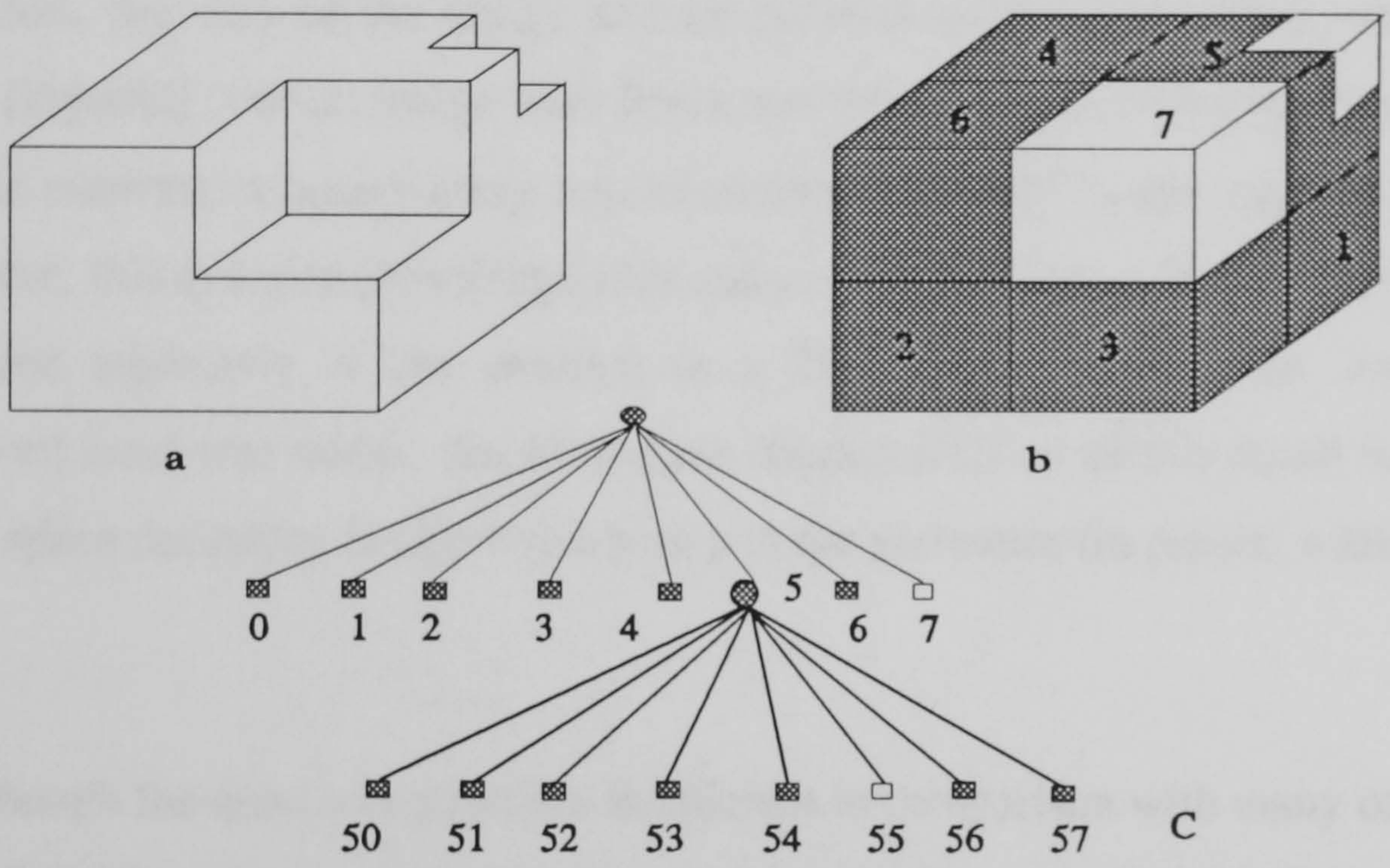


Figure 2.4 (a) An object, (b) Decomposition of object, (c) Oct-tree structure.

Figure 2.4 shows an implementation of the oct-tree structure. The resolution of both the quad-trees and oct-trees is defined as the number of times that the decomposition process is applied. The resolution may be fixed beforehand, or it may be governed by properties of the input data.

Quad-trees and oct-trees have been applied to computer graphics, image processing, cartography, solid modelling and related fields [Samet90]; they have also been used to represent point data, regions, curves, surfaces, and volumes. In this thesis, quad-tree and oct-tree approaches are primarily used to organise regions of terrain elevation data into an efficient format for real-time aircraft navigation usage.

2.3 Quad-tree Data Structures

2.3.1 Space Efficiency Considerations

A prime motivation for the development of the quad-tree is that it offers a reduction in the amount of space necessary to store data by representing homogeneous blocks as single items. The amount of space required for a quad-tree is a function of the resolution, the size of the image and its position in the grid within which it is embedded [Dyer82]. For an image with Black and White blocks, $4(B+W)/3$ nodes are required. In contrast, a binary array representation of a $2^n x 2^n$ image requires only 2^{2n} bits; however, this quantity grows logarithmically. Dyer has shown that a square of size $2^m x 2^m$ placed arbitrarily at any position in a $2^n x 2^n$ image requires an average of $O(2^{m+2}+n-m)$ quad-tree nodes. An alternative characterization of this result is that the amount of space necessary is $O(p+n)$ where p is the perimeter (in pixels) widths of the block.

Although the quad-tree structure is efficient in comparison with many other data structures including arrays, lists, queues, and run-length segments, it also generates considerable storage overheads. The main disadvantage of this structure is the high internal storage cost associated with the storage of five pointers per node. The size of these pointers must be sufficiently large to address any location within a data base. On the other hand, only one field is used to store the description of the nodes. The memory needed to store pointers is thus far greater than the memory required to store node information. Furthermore if the amount of aggregation is minimal (e.g. a checkboard image), the quad-tree is not efficient. Consequently, considerable attention has been given to the development of pointerless quad-trees [Garg82, Mark85].

2.3.2 Pointerless Quad-trees

A number of quad-tree representations have been developed which do not use pointers to organize the nodes [Garg82, Mark85, Lauz85]. The basic idea is that each

leaf node is encoded by an integer number, termed a *locational code* or *key* corresponding to a sequence of digits, where the value of each digit reflects successive quadrant subdivisions and locates the leaf position from the root of the quad-tree. Various formats of locational codes adopted for pointerless quad-trees define the variants of quad-trees.

For example, in Figure 2.5, a sequence (0 1 2 3) of values represents the four leaf nodes of a single subdivision region, given in an ordered set (NW NE SW SE) by convention. As the NE quadrant is not a leaf node, it can be further subdivided, then this region can be written either in a depth first format by storing the data as (0 1 10 11 12 13 2 3) or a breadth first scheme as (0 1 2 3 10 11 12 13).

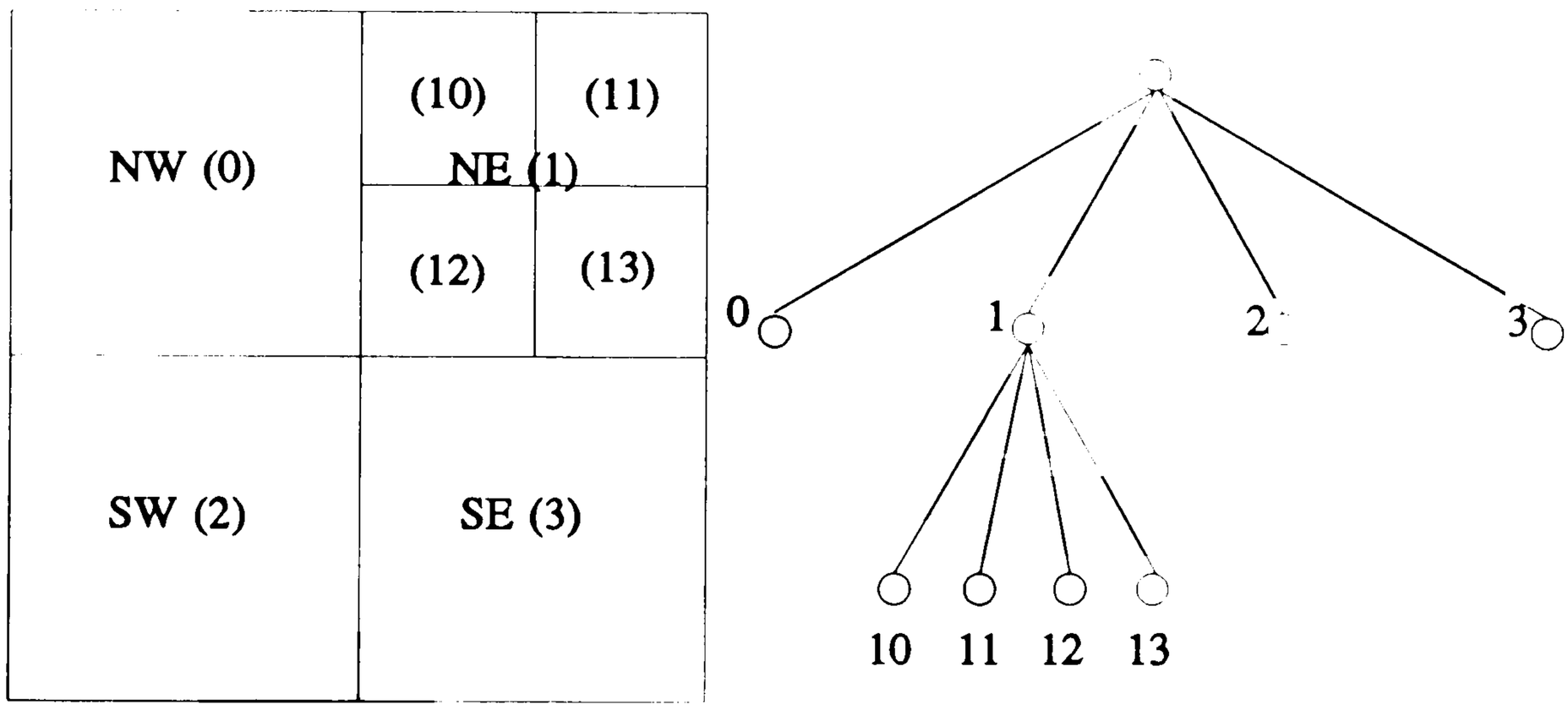


Figure 2.5 An example of quad-tree region subdivision representation.

The earliest concept of a locational code was used as an index to a geographic data base by Morton [Mort66] and is termed the *Morton numbering sequence*. The Morton sequence is defined as the assignment of consecutive numbers to the cells in a grid, beginning with 0, such that in the number's binary representation, the odd-numbered bits denote the co-ordinate position along one axis and the even-numbered

bits denote the co-ordinate position along the other axis, where the co-ordinates begin at (0,0). Figure 2.6 shows an example of the Morton numbering system.

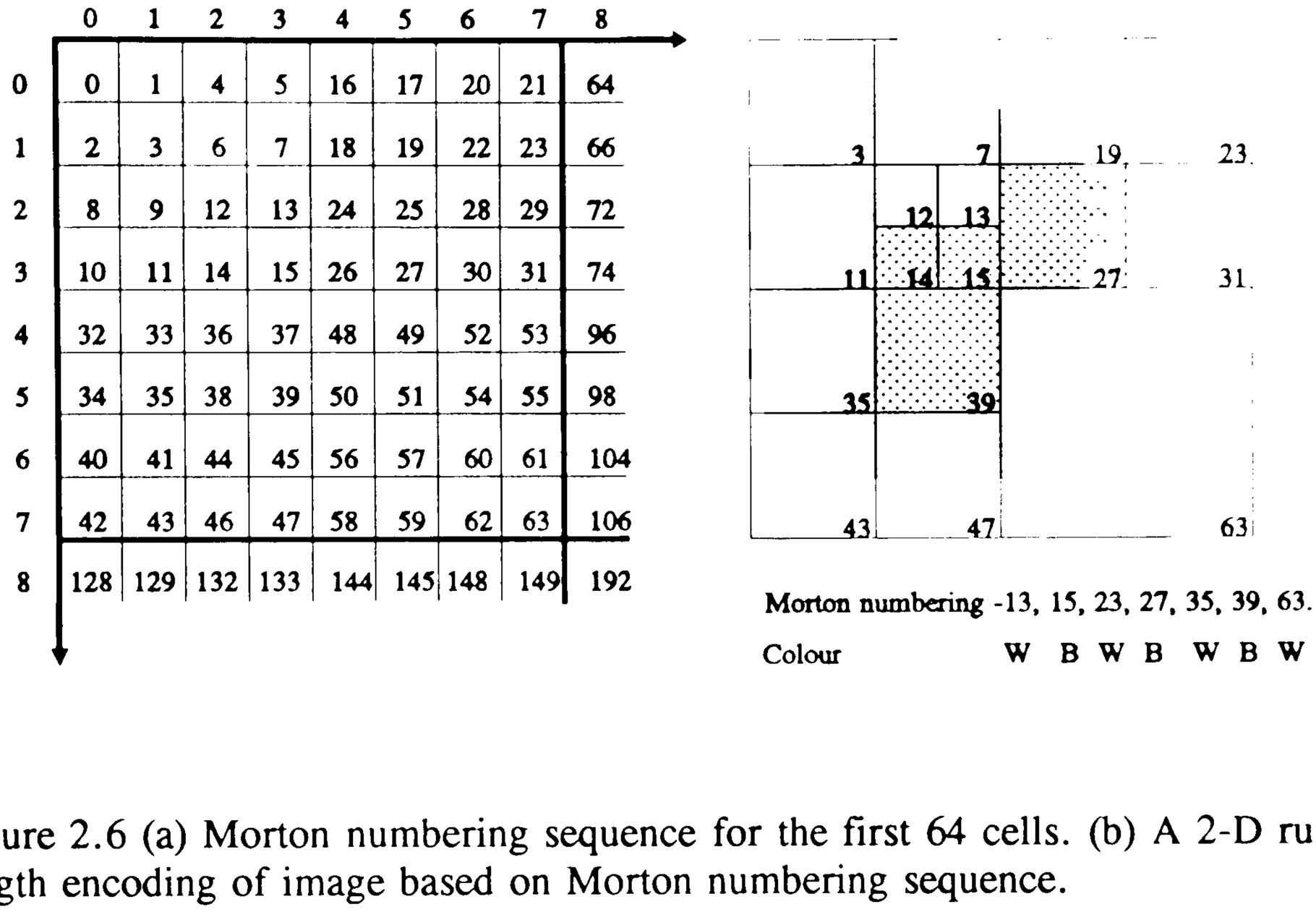


Figure 2.6 (a) Morton numbering sequence for the first 64 cells. (b) A 2-D run-length encoding of image based on Morton numbering sequence.

The Morton numbering sequence has been widely used as the index of the leaf nodes of quad-trees [Klin79, Garg82a,c, Sam90]. Usually the collection of leaf nodes is kept as an ordered list that is stored according to the value of the locational codes of the nodes. In many applications, this list is implemented as a tree structure to improve the speed of search and retrieval operations. A major disadvantage with this approach is the loss of flexibility when traversing the quad-tree. In other words, the quadrant nodes can only be visited in the order in which they are initially stored.

2.3.3 Variants of Linear Quad-trees

When a list only contains locational codes for black nodes it is termed a *linear quad-tree*. The linear quad-tree, introduced by Gargantini [Garg82a], uses a locational code to represent each BLACK leaf node and ignores the grey and white nodes in the

tree. This locational code is analogous to taking the binary representation of the interleaved values of the x and y co-ordinates of a designated pixel in a block and interleaving them (as shown in Figure 2.7). This "linearizing" mapping preserves the spatial locality of the original two-dimensional image in one dimension.

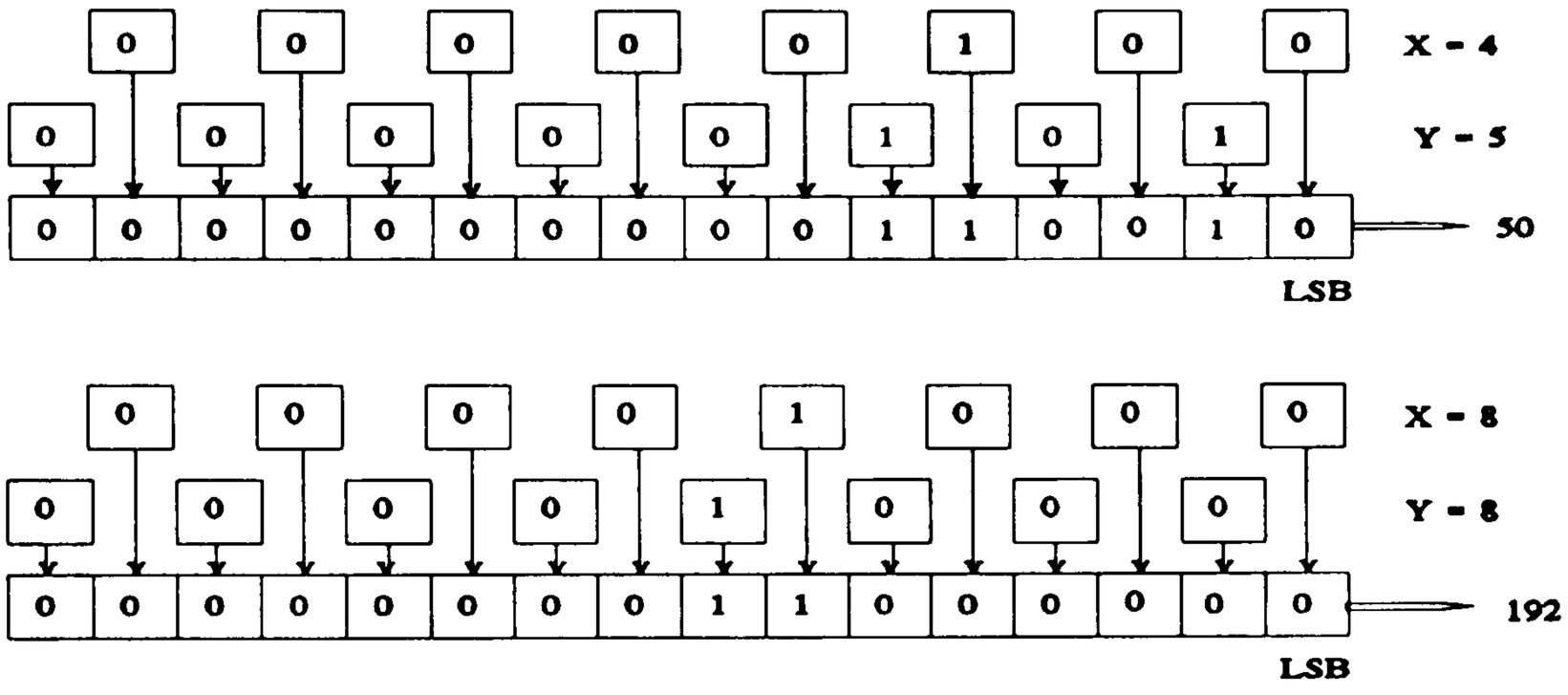


Figure 2.7 Binary representation of co-ordinates (X,Y) interleaving.

For a quad-tree of depth n-1, a node in level m is represented by the sequence of n digits $<q_{n-1},q_{n-2},\ldots,q_0>$, given by relation 2.1 (where n=root, m=node's level, m=0 for pixel level). This sequence forms a number Q, given by the relation 2.2. Each digit q_i represents a subdivision code that can take on a value of 0=NW, 1=NE, 2=SW, 3=SE, or 4=MERGED. The number of digits (code length) is fixed to the level of the quad-tree (as shown in Figure 2.8d).

$$q_i = \begin{cases} 4 & 0 \le i < m \\ \text{quadrant type}(P_i) & m \le i < n \end{cases} \qquad 2.1$$

$$Q = \sum_{i=0}^{n-1} q_i \, 5^i \qquad 2.2$$

The resulting linear quad-tree consists of a sorted list of the locational codes of BLACK leaf nodes, corresponding to the order in which the leaves are encountered in a post-order traversal of the corresponding regular quad-tree.

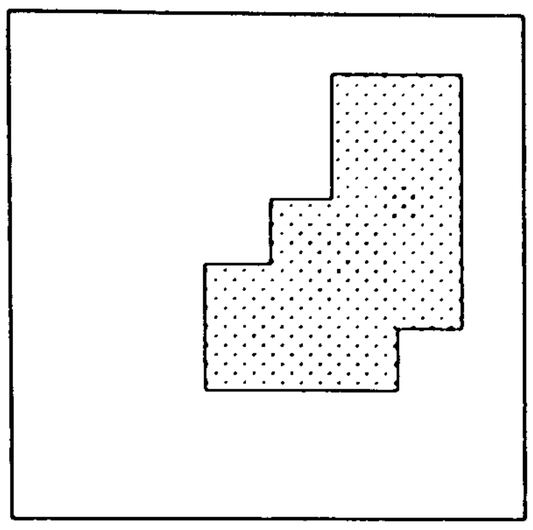
Abel and Smith [Abel83] report an almost identical encoding; the only difference being that the roles of digits 0 and 4 are interchanged, that is, 0 is used to denote the size of the nodes. A drawback of these locational codes is the use of base 5 numbers to represent the codes of subdivisions implying the use of division operations instead of shift and modulo operations when performing the decoding process.

This problem is overcome by Gargantini who demonstrated a variant of these locational codes using a 2 bit digit to represent each quadrant code combined with the level information of the nodes [Garg82c], without using 4 or 'X' to mark merged nodes. Assuming an image of size 2^n x 2^n, the locational codes of each leaf node of size 2^k x 2^k is n digits long; the leading n-k digits contain the subdivision codes that locate the leaf node along a path from the root of the tree. The k trailing digits contain the subdivision codes that locates the pixel in the corner of 2^k x 2^k block. Figure 2.8e shows this variant of linear quad-tree.

Samet also described how locational codes can be computed directly from an explicit pointer based quad-tree representation and also to reconstruct the explicit quad-tree from the locational codes of leaf nodes [Same90]. This is a conversion process between pointer-based quad-trees and key-based quad-trees. Each node at level m in a quad-tree, where level n is represented by the sequence of n - m digits $<q_m..,q_{n-1}\ q_n>$, corresponds to the path from level n to m, where 1=NW, 2=NE, 3=SW, 4=SE. The sequence also forms a number Q, given by equation 2.3. The resulting locational code has a variable number of digits according to the size of the node. The resulting list of variable length locational codes is termed a VL (Variable length) linear quad-tree (shown in Figure 2.8f). Samet refers to Gargantini's approach as FL (fixed length) linear quad-trees.

$$Q = \sum_{i=0}^{n-m-1} q_i\ 5^i \qquad 2.3$$

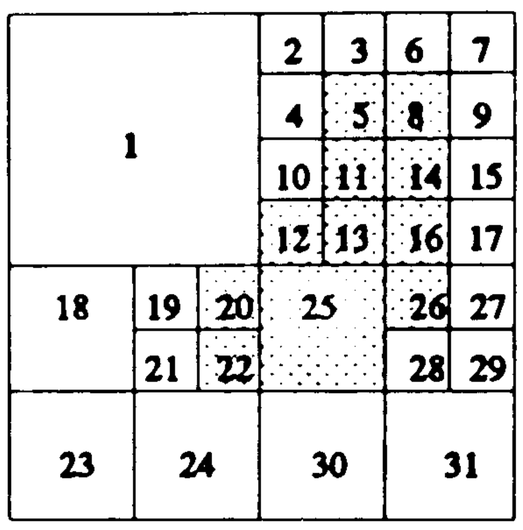
where $1 \le q_i \le 4,\ 5^{n-m-1} \le Q \le 5^{n-m}$.



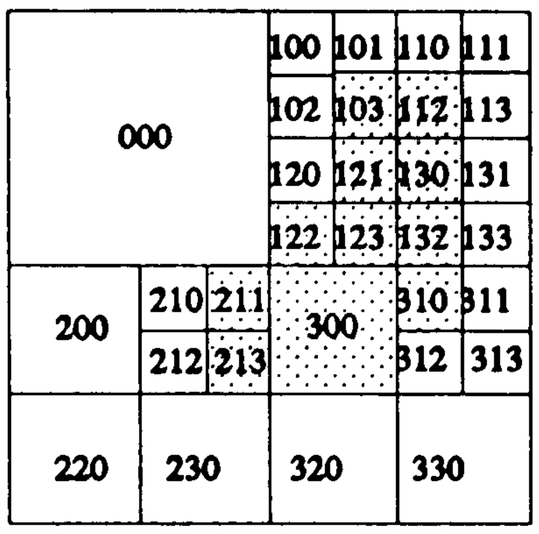
(a) a binary image.

0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	0
0	0	0	0	1	1	1	0
0	0	0	1	1	1	1	0
0	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

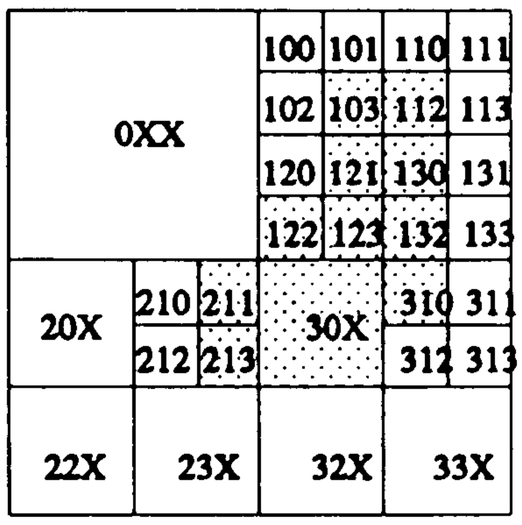
(b) binary array



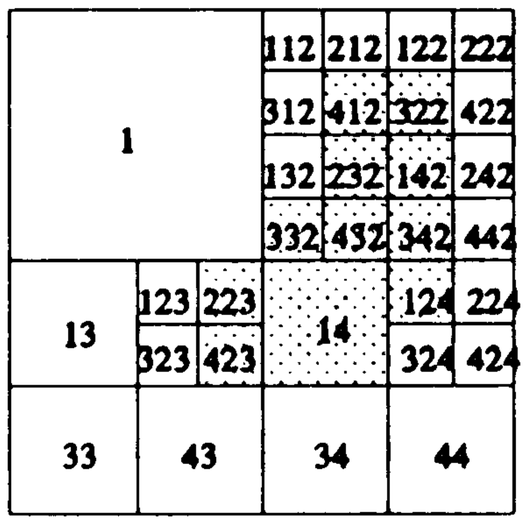
(c)nodes sequence in quadtree



(e) FD locational codes of quadtree.



(d) FL locational codes of quadtree.



(f) VL locational codes of quadtree.

(g) linear quadtree structures, only black nodes are stored.

Figure 2.8 A binary image, its corresponding binary array and various quad-tree representations.

When VL locational codes (representing the black nodes) are sorted in increasing order, the resulting sequence is a variant of the breadth-first traversal of the black nodes, as shown in Figure 2.8f. The breadth-first property of the VL locational code means that the successive nodes provide a better approximation.

When applying the Morton numbering sequence to the locational codes of a quad-tree, the cells within any of the subquadrants at any level of the quad-tree have consecutive numbers. Lauzon's *two-dimensional run-encoding* (2DRE) [Lauz85] is a technique for compacting a linear quad-tree by exploiting this property. He proposed that whenever the linear quad-tree has consecutive leaves of the same colour, only the key of the last leaf in such a "run" is stored. This key is the Morton number of the pixel that occupies the lower right corner of the leaf node. The list of leaf nodes is then sorted by their corresponding Morton numbers. For example, Figure 2.6b depicts an 8x8 image, where the 16 nodes of the quad-tree are sorted in increasing order of their Morton numbers : 3, 7, 11, 12, 13, 14, 15, 19, 23, 27, 31, 35, 39, 43, 47, and 63 respectively. After discarding all but the last element of adjacent nodes of the same colour, the process yields a list of 7 nodes with corresponding Morton numbers 13, 15, 23, 27, 35, 39, 63.

Gargantini showed that in a regular quad-tree of N nodes, the best case and worst case number of nodes in the corresponding linear quad-tree are 0.17N (one Black node for every three white nodes) and 0.54N (three BLACK nodes for every WHITE node) respectively [Garg82a].

The 2DRE method usually reduces the number of records to about 1/2 to 1/3 of the number of leaf nodes, and is more compact than linear quad-trees [Mark85]. However, the 2DRE method is restricted to representing only region data (run-encoding in Morton numbering sequence). Since the decomposition rules require that two entities are separated, such data as points and lines are recursively partitioned until each quadrant is unity. Clearly points and lines are not applicable to run encoding.

At present, the linear quad-tree is the most widely used quad-tree structure, primarily because of the reduced storage it offers for quad-trees. The availability of efficient schemes for external storage management has also been a factor in the widespread use of the linear quad-tree.

Although the pointer based quad-tree has a pointer system and a good hierarchical structure, it is inappropriate for applications dealing with large image data. If quad-trees representing an image are too large to fit into the internal memory, then they must be stored in the external memory and consequently the nodes of the quad-tree may be stored over several pages. Accessing pointers becomes much more costly because reference to a node on another page usually requires a disk access.

Furthermore, a large quad-tree is difficult to manage on secondary storage. Consider the file management system in GIS as an example. GIS are intended to handle very large, multi-layer heterogeneous spatial indexed data [Mark86, Meno87, Shaf89]. The amount of data is so large that it is impossible to store the whole data base in main memory. File management techniques must be employed so that at any instant of time, only a portion of the data base is resident in the main memory and the rest of the data base is stored in external storage as a set of files.

The most commonly used file management method for linear quad-tree-based GIS is the B^+-tree [Come79]. A B^+-tree is an indexing method for a very large file of a sorted list. Abel has presented a B^+-tree structure that use the values of the nodes of a linear quad-tree which are arranged in ascending order as the key to access the quad-tree from secondary storage [Abel84]. A number of GIS based on the linear quad-tree model and based on the B^+-tree file management scheme are currently in use [Abel85, Mark86, Shaf89].

2.4 The Construction of Quad-trees

The quad-tree is proposed as a representation for binary images because its hierarchical properties facilitate a large number of geometric operations, including union, intersection, complement, transformation, point location, overlay, search, adjacency finding, and windowing.

However, most images are traditionally represented by structures such as binary arrays, rasters, chain codes (also termed boundary codes), or polygons (vector, edge, vertices). Therefore techniques are required that enable quad-trees to be efficiently constructed from these various representations and readily switched between these forms. Such algorithms includes quad-trees extraction from binary arrays [Same80a], conversion between quad-trees and boundary codes [Same80b,84b, Latt91], and quad-trees derived from vector representations of polygons [Mark85]. Since the terrain is an area data representation, most attention has focused on the conversion of region data.

Gargantini's algorithm encodes each pixel from a binary array into its quaternary code [Garg82a,83]. The quaternary code can be obtained by taking the binary representation of the interleaved values of the x and y co-ordinates of a pixel. If the encoding process is performed in row sequence, the list is sorted after all the black pixels are encoded and a reduction process is recursively applied to merge four blocks belonging to the same quadrant. If the encoding of pixels is performed in a Morton number sequence, the reduction process can be applied directly to obtain the maximal blocks.

Lauzon's 2DRE algorithm is based on the assignment of a Morton number to each cell in an image, and it uses these Morton numbers to encode maximal leaves; the resultant file is a sorted list of leaf nodes. The value associated with a particular location is given by the smallest Morton number in the file which is greater than or equal to the Morton number of the location [Lauz85].

Samet's pointer based quad-tree construction algorithm [Same80a] only examines each pixel in the binary array once in a manner analogous to a post-order tree traversal. The execution time of this method is proportional to the number of pixels in the image.

An image can also exist as a sequential file of consecutive rows. Such a representation is termed a *raster* representation and consists of a list of the pixels ordered by rows. The algorithm for converting a raster image to a quad-tree rearranges each pixel of the raster image to be inserted into the quad-tree in raster order. As the quad-tree is constructed, nodes are merged to yield maximal blocks [Same81].

Basically, the oct-tree structure for the representation of 3-D objects is an extension of the quad-tree representation of 2-D (binary) image. Chen and Huang made a survey on the construction of oct-trees according to the formats of the input data [Chen88]. They classified the input data into five categories : 3-D arrays, quad-trees of serial section, silhouette images of an object, images of the depth of an object and object models. They also observed that most quad-tree construction algorithms, using the 2-D binary array discussed above, can be directly extended and applied to 3-D cases.

2.5 Adjacency and Neighbour Finding

The nodes being processed in a quad-tree correspond to the blocks in a region. The spatial adjacency relationships between blocks play an important role in quad-tree structures. There are several applications where there is a need to locate neighbouring pixels efficiently :

- Locating the pixel or block adjacent to a given pixel or block in a specified direction to determine the boundary of a region or to determine the connectivity [Garg82a].
- Reconstruction of a region quad-tree from its locational codes of contour pixels or nodes [Same80b,89].
- Checking the adjacent or neighbouring nodes around a set of boundary pixels to

obtain the quad-tree [Latt91].

The neighbours of a block can be of different size, position or adjacency. In quad-trees, adjacent nodes can be along 4 edges or along 4 vertices. In oct-trees, adjacent nodes can be along 6 faces, along 12 edges, or along 8 vertices. To specify a neighbour, precise information is necessary about its size, location and whether it is a leaf node.

For example, it is necessary to be able to distinguish between neighbours that are adjacent along an entire edge of a node and those that are adjacent along only a segment of a node's edge. The difference implies that the size of a corresponding neighbour is equal to or smaller than a given node. The same distinction can also be made for vertex and face directions.

Generally, functions are defined that express these relations precisely [Same82, 89]. For example, a function yields the quadrant value of equal size that shares the edge or vertex of a given quadrant; a function of table checking gives the result 'true' if, and only if, a quadrant is adjacent to a specific edge or vertex of a given quadrant.

The methods of locating neighbours in both pointer and pointerless quad-tree are analogous; the main difference is that the bit manipulation operations in pointerless methods [Garg82a] is substituted for pointer tracing in pointer-based methods [Same82]. The most general pointer method termed "Nearest Common Ancestor Method" is based on locating a nearest common ancestor of a node and its possible neighbour in a given direction [Same82,89]. This method uses the four links from a node to its four sons and one link to its parent for a non-root node.

The approach is to ascend the quad-tree until a common ancestor is located and then descend back down the quad-tree in search of the neighbour node. Tables of quadrants, edges and spatial relationships of vertices representing the blocks are predefined and used to locate the proper node of a neighbour. When this method is

applied to a key-based quad-tree, the neighbour-finding search is performed by computing the path component of locational code of reflection until the nearest common ancestor is reached, followed by a search to determine if such a node path actually exists in the nodes list.

Gragantini's algorithm [Garg82a] for determining the neighbour of a given node in a linear quad-tree alters the locational code of the query node by changing the appropriate digits of the code to match the location of the desired neighbour. A binary search is then applied to the sorted list of quad-tree nodes. Neighbour finding is an important technique in the efficient implementation of a number of algorithms that use quad-tree and oct-tree representations, such as connected component labelling [Garg82b], the computation of geometric properties [Garg83, Same85], and the location of points and objects in an image.

2.6 Quad-trees and Related Terrain Representations

Unlike a binary image, terrain elevation data contains multi-valued data of each point within a specified lower and upper limit of the overall range of elevation. The pointer based quad-tree can be extended from the binary to the multi-valued case. The value field in a non-leaf node of a regular quad-tree may be used to store an appropriate function of the values in the corresponding portion of the array. Each leaf node represents a quadrant with a uniform value or colour, which is stored in its value field.

Linear quad-trees are by definition lists of BLACK nodes and contain no representation for internal nodes. They are not extendable to multi-valued cases. An alternative for linear forms of quad-trees for GIS applications is to use a collection of binary quad-trees to represent elevation data, one for each elevation range of terrain [Meno87]. The topographic component thus consists of one binary quad-tree for each range.

Since neighbouring grid cells seldom have identical values in terrain elevations, Cebrian proposed a method for the representation of a terrain surface [Cebr85]. He observed that it is highly desirable to represent the topography as precisely as possible within a GIS and converted the pixel ordering from the raster (row by row) to the Morton numbering sequence, where the Morton sequence is used to provide cell addresses in the terrain data file. This scheme stores all the lowest leaves of a quad-tree but omits all nodes at higher levels of the tree. Thus the terrain surface is a linear quad-tree which is compatible with a binary image. However, this approach loses the hierarchical properties of quad-trees and introduces redundant information.

Lauzon also proposes a file structure in which the Morton number of a cell provides the address of the storage location for the elevation [Lauz85]. Chen uses a mathematical approximation method that recursively subdivides the data domain into four quadrants [Chen86]. The elevation data within each quadrant is replaced by an approximate mathematical surface. It means that the values of the pixel within the quad-tree node are represented as a two-dimensional surface F(i,j). If the absolute difference δ between F(i,j) and a mathematical function f(x,y) (where $0 \leq x,y \leq 1$) is less than a given error range ϵ, then F(i,j) can be approximated by f(x,y). Each node is represented by the address of the quad-tree node in Morton numbering and its elevation given by the function f(x,y).

The surface approximation functions which had been evaluated are planes (average, maximum, minimum), ruled surfaces and quadric surfaces. Chen has shown that as the given error range δ decreases, the quad-tree size increases. However only the one surface of the maximum plane value can be applied to an airborne environment.

In the design of a terrain model for real-time airborne applications, it is required that a terrain model is not only a compact hierarchical data structure, but also an efficient addressing strategy in which elevation data of a given position can be easily accessed and its spatial adjacency can be readily located for further processing.

Both Chen's and Cebrian's methods partially meet the above requirements. They all use the Morton number as a storage address of elevation in quad-tree nodes. The difference is that Chen uses mathematical functions to maintain the hierarchical properties of a quad-tree [Chen86] with the computation cost, whereas Cebrain keeps the nodes at the lowest level of a quad-tree in terms of its topography [Cebr85] but without significant saving in storage.

In chapter three, a variant of the 3-D locational code notation is used to represent a terrain model instead of using a 2-D locational code with an extra field for elevation data. A scaling function is adopted to approximate the terrain elevation data for the merging process during the construction of a terrain oct-tree.

2.7 Path Planning Preliminary

2.7.1 Navigation Environment

A major objective of aircraft navigation is to select a flight path to enable the aircraft to travel from a given start point to a goal point. If a designated navigation area is given, the path planning for aircraft navigation can be performed prior to take-off. While an aircraft is navigating in a hostile area, the flight path needs to be modified dynamically to respond to new threats and changing obstacles in the environment. In order to execute navigation in such demanding environments, an aircraft will require continuous access to the terrain data for real-time path planning.

Flight path planning is analogous to robot motion planning, which consists of locating obstacles and checking collisions as well as collision avoidance. In robot motion planning, the navigation environment can be classified into two categories: the known environment and the unknown environment [Schw88].

In a known environment, the geometry is known to the robot. The robot motion planning problems involve finding a collision-free path for a moving object among a set

of known obstacles. Knowledge of the navigation environment leads to different approaches to constructing a search space to find a path.

For example, several approaches [Udup77, Loza83, Broo83] assume a complete prior knowledge of the navigation space; i.e., the dimensions, positions, and orientations of all the obstacles in the navigation space are known. In these approaches attentions is given to the modelling of obstacles.

In an unknown environment, the geometry of the environment is not fully known to the robot system, therefore a robot needs to be able to sense its local environment to determine where it is, where it needs to go and what obstacles are in its way. Once the robot determines this, it then generates a path that will get it from its current location to its destination while avoiding the obstacles in its way [Loza83].

Environment sensing and path planning are two distinct tasks that constitute robot motion planning. Many autonomous land vehicle path planning algorithms [Mitc84, Dent84, Keir84] rely both on a digital database and sensor-based information to plan a route. The sensory data is incorporated into a global path in order to achieve real time local path obstacle avoidance. In TRN navigation, the sensing process is incorporated into the accessing of a terrain data base for position checking or collision avoidance [Prie90].

2.7.2 Path Planning Approaches

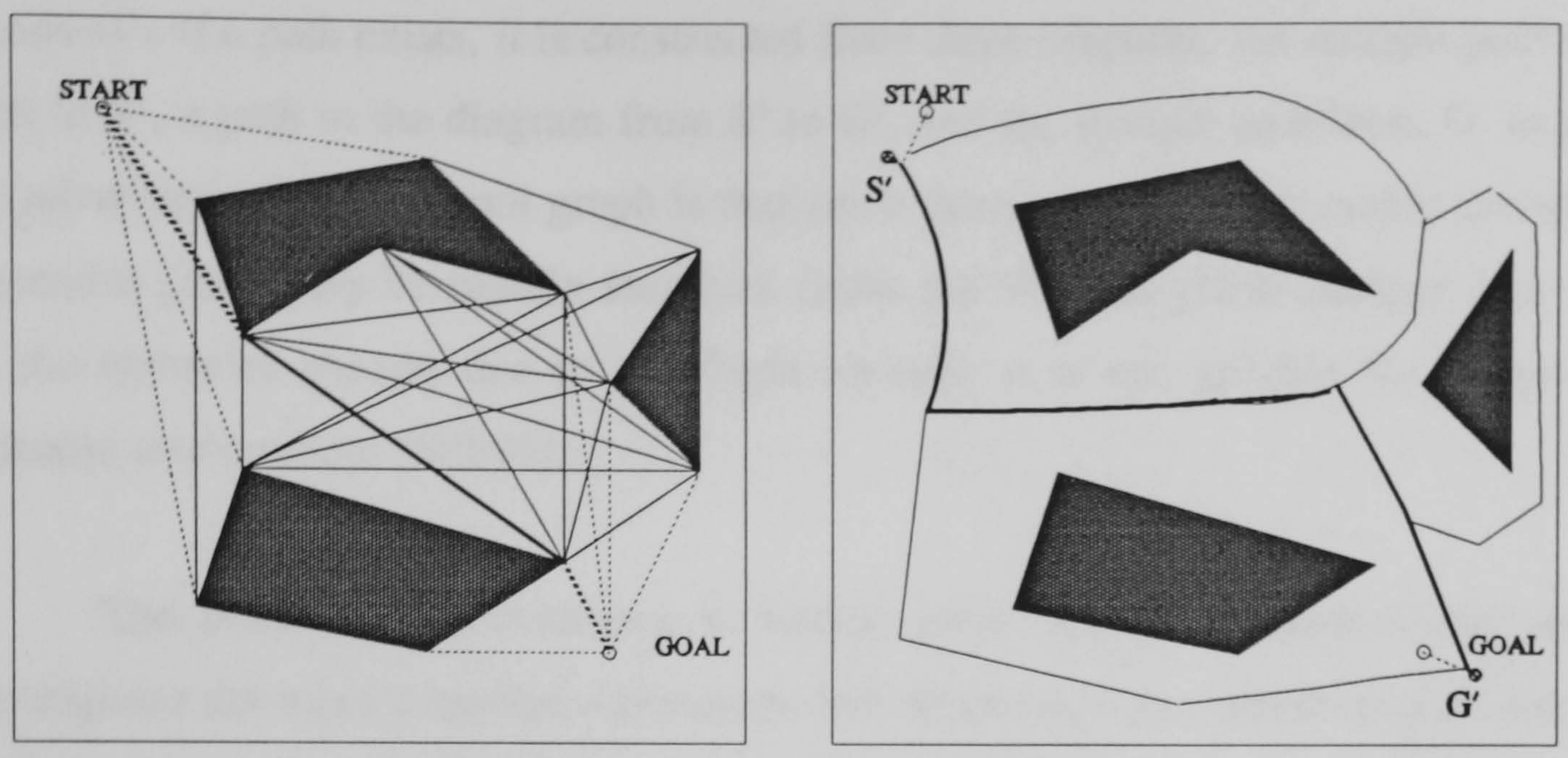
A large number of methods has been applied in solving basic motion planning problems. Usually, the robot is small in comparison with the dimensions of the terrain map and can be modeled as a point [Udup77]. Lozano-Perez and Wesley apply this idea to motion planning, where many problems of moving a robot among obstacles can be reduced to the case of moving a point, by mapping the obstacles into a form that describe the locus of forbidden regions of the reference point of the robot [Loza79]. The basic motion planning problem is then defined as follows:

Let B be a single rigid robot, which is represented as a point in the robot's *configuration space* [Loza79] (which is synonymous with *navigation space* in this thesis). The obstacles are mapped to this configuration space, where a *configuration space* is defined as a two or three dimensional space V amid a collection of obstacles whose geometry is known to the robot. Given an initial position START and a desired final position GOAL of B, it is necessary to determine whether a continuous obstacle-avoiding motion of B exists from START to GOAL, and if so, to plan such a motion. The problem is equivalent to the problem of calculating the path-connected components of the *free space* of B, where *free space* is defined as the set of free positions of B in which B does not contact any obstacle.

The approach to path planning consists of obtaining the connectivity of the robot's free space in a network of one-dimensional curves, called the *roadmap,* lying in the free space [Lato91]. Once a roadmap has been constructed, it is used as a search space consisting of all the possible paths that will be considered. Path planning is thus reduced to connecting the start and goal points by points in the roadmap. The three most common ways to build a roadmap - *visibility graph, Voronoi diagram* and *grid graph* are as follows:

The *visibility graph method* mainly applies to two-dimensional space within a polygonal obstacle region [Loza83]. The visibility graph is the non-directed graph whose nodes are the start and goal points, and all the obstacle vertices. The links of this graph are straight line segments connecting two nodes that do not intersect the interior of the obstacle region. Figure 2.9a shows an example of a visibility graph.

As the size of a visibility graph is determined by the number of vertices of obstacles [Mitc88], it is an excellent choice for path planning problems where the number of obstacles is small and static. Since the visibility graph changes over time as the obstacles change, it is not possible to build a static graph and use it in a dynamic environment [Silb91].



(a) Visibility graph

(b) Voronoi diagram

Figure 2.9 The visibility graph and Voronoi diagram in a 2-D space with obstacles.

Another roadmap method, termed *retraction*, introduced by O'Dunlaing and Yap, proceeds by retracting the configuration space of a robot onto a lower-dimensional subspace (the roadmap) [O'Dun82]. In two-dimensional configuration space, free space is typically retracted onto its *Voronoi diagram*. This diagram is the set of all the free space whose minimal distance to the obstacle region is usually achieved with at least two points in the boundary of obstacles. This minimum distance ensures a collision free path to a robot [O'Dun82].

When the obstacles are polygons, the Voronoi diagram consists of straight and parabolic segments which are the locus of free space [Meng88]. A *Voronoi graph* is then made by placing a node at every "T" intersection of the corresponding Voronoi diagram. A "T" junction is simply where three lines meet. After a Voronoi graph is created, the graph is searched for a path.

For instance, the Voronoi diagram in figure 2.9b is searched for a path between S' and G'. If a path exists, it is constructed from three subpaths: the straight path from start to S', a path in the diagram from S' to G', and the straight path from G' to goal. An advantage of the Voronoi graph is that since there are only a few nodes generated, the entire graph may be rapidly searched. Since the Voronoi graph changes over time as the obstacles change due to the flight altitude, it is not suitable for a dynamic airborne environment [Silb91].

The third method described is termed *grids*. A cell decomposition method decomposes the robot's configuration space into simple regions, called *cells*. A cell that contains any underlying obstacle is impassable. A non-directed graph representing the adjacency relationship between the non-obstacle cells is then constructed and searched. This graph is called the *connectivity graph* or *grid graph* [Lato91]. Its nodes are the non-obstacle cells and two nodes are connected by a link, if and only if, the two corresponding cells are adjacent.

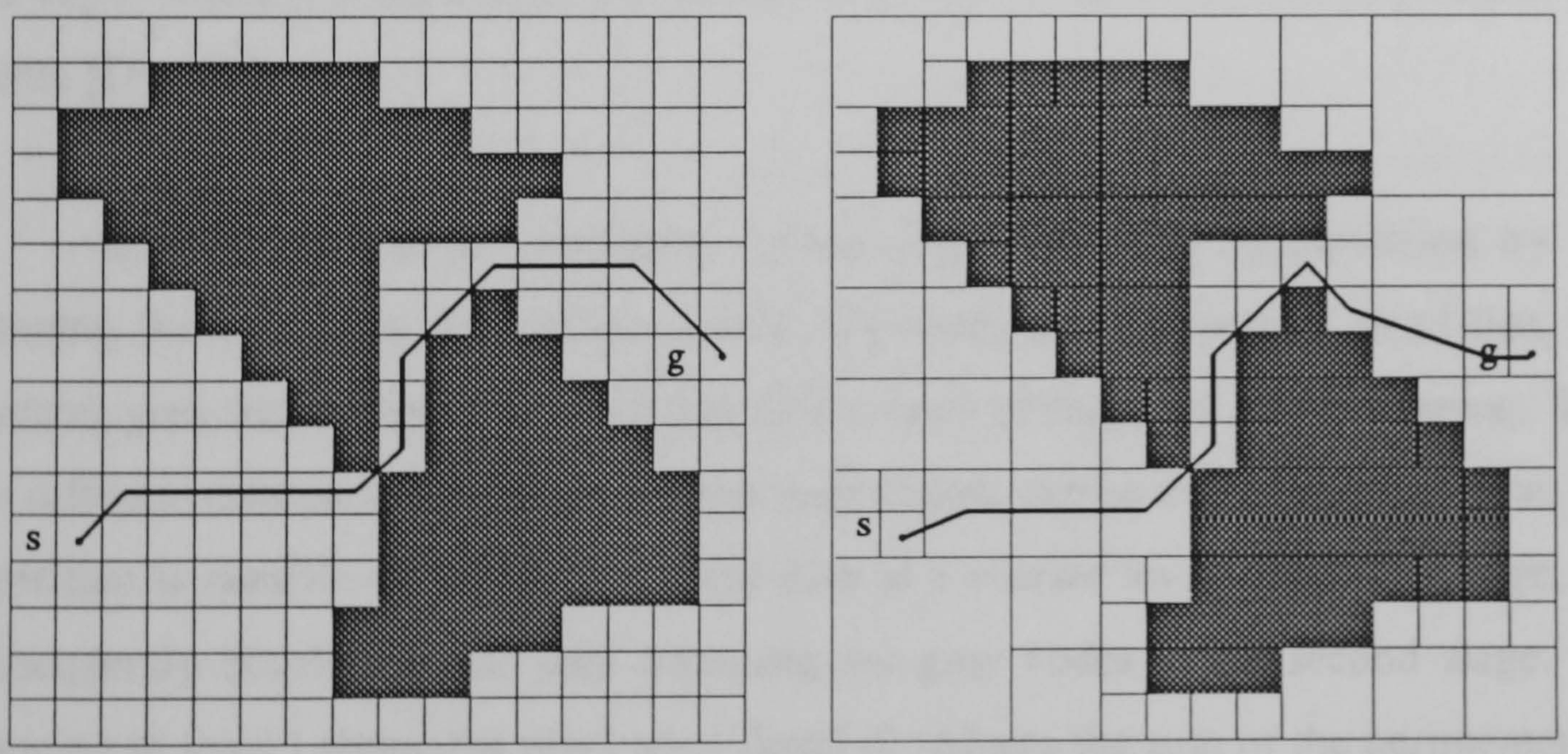


Figure 2.10 Examples of cell decomposition.

For instance, the decomposition process subdivides the configuration space into small equal sized squares and each square is labelled as either an obstacle or a non-obstacle [Mitc88]. The result is a grid tessellation similar to overlaying a two dimensional binary array over the configuration space as shown in Figure 2.10. The search for a path is performed in a graph whose nodes are the free space squares and with edges connecting adjacent squares. Paths are found by moving through adjacent cells of the grid that are passable until the cell containing the destination is reached.

For a very large number of cells, the search time may become prohibitive. One method of reducing the number of cells, is to exploit the hierarchy of the search space and quad-trees offer a solution to this problem. In a quad-tree representation, if there are large areas of free space or obstacles, then those areas can be represented by a few large blocks in the quad-tree and can be dealt with as units by a path planning algorithm. The speed of the path searching process is increased by searching the nodes in the quad-tree instead of searching the grid points in the configuration space, and the search can be performed in any desired resolution level of the quad-tree. In fact, the number of leaf nodes in a quad-tree having polygonal obstacles is approximately $2/3*O(p)$, where p is the sum of the perimeters of the obstacles in term of pixels or grid points [Dyer82].

Kambhampati et al. improved a quad-tree-based planning algorithm by first planning the path in a reduced-resolution quad-tree, called a *pruned* quad-tree, that contains grey leaf nodes, corresponding to mixtures of obstacles and free space. They use different criteria to determine whether a grey node can be treated as free space. The algorithm is capable of planning a global path at a coarser level in the first stage, and subsequently developing the path accessing the grey nodes in the second stage. The planning in level l above the pixel level (level 0) reduces the sum of the perimeters and the number of leaf nodes by a factor of 2^l, thereby substantially reducing the time complexity of the path search process [Kamb86].

Applying the grid method to three-dimensional problems involves the use of three-dimensional grids. All other aspects of the method remain the same. When three-dimensional problems are addressed, the number of cells drastically increases. As a result, path planning algorithms based on the grid method require both large memories and high processing capability.

2.8 Terrain Modelling for Navigation

An algorithm to find paths among obstacles depends on the configuration space (or navigation space) representation. From data structure considerations, two standard approaches are used to model the terrain for obstacle avoidance [Mitc88]:

- Digitized models - in which the navigation space is subdivided into small regular cells, and each cell is labelled as an obstacle or a non-obstacle (free space). In models such as binary grids and quad-tree structures, each obstacle is represented as BLACK (1) or black node, where free space is WHITE (0) or white node.

- Geometry models - in which an obstacle or a non-obstacle is defined by a given description of a polygonal space in terms of its boundary representation. Usually, obstacles are defined as a list of simple polygons, each represented by a doubly linked list of vertices, and each vertex given by a pair of co-ordinates. The vertices of obstacles are used to form the visibility graph which is constructed by connecting every pair of vertices by a straight segment, provided this segment does not traverse the interior of any obstacle [Loza79].

For airborne applications, two reasons make the digitized model difficult to implement. Firstly, the path is found by expanding the path segment from the current point to the neighbouring free point, although these stepwise movements [Kamb86] of a local path are not suitable for high speed aircraft manoeuvres. Secondly, the path planning algorithm must search and expand a large number of cells during its execution

which makes the search process unsuitable for real-time operation. The path planning cost increases with grid size rather than with the number of obstacles present.

Lozano-Perez observed that the most important factor for a path-planning space representation is to avoid excess detail on parts of the space that do not affect the planning operation [Loza81]. Hierarchical structures such as quad-trees can avoid much of the expansion by aggregating the region of free space and the region of obstacles to areas of larger squares, in order to reduce computational cost. However, the large number of nodes and the stepwise movements of local paths are still unacceptable in an airborne environment that needs to continually update the planned path, because the aircraft traverses it, as new obstacles are formed at various altitudes.

In a geometry model, the obstacles that constitute the visibility graph are formed in a list of polygonal obstacles [Loza79]. This representation of obstacle space is static with the exception of moving obstacles. However, in aircraft path planning, the search space can not be treated as static as the navigation space and obstacles will change according to the flight altitude. The visibility graph must be reformed according to a new set of polygon obstacles. Unless the geometry of obstacles are predefined for each possible altitude (also termed flight altitude in these systems), the obstacles in the navigation space need to be explored.

The exploration process makes substantial changes to a representation (visibility graph, Voronoi diagram, etc.,), which is easier to analyze before planning the path. A potential practical shortcoming of this approach is that the path planning cost is very high because of the representation conversion process involved [Kamb86]. It is difficult to provide this type of obstacle information dynamically in a real-time environment.

Since digital terrain elevation data has been used extensively in airborne navigation, an efficient method is needed to reorganize the large amount of terrain data and to be able to update the navigation space quickly for global path planning. A dynamic approach should be employed in which the flight plan generation is tightly

coupled to the gathering of obstacle data from the navigation space, which in turn, changes rapidly according to flight conditions. For very large databases used to represent airborne terrain, the quad-tree structure provides a 'pruned' terrain database to speed up the planning process.

In the following chapters, a global and dynamic flight path planning algorithm is developed, based on oct-tree terrain representation. The path planning algorithm demonstrates that the oct-tree model is capable of real-time aircraft navigation applications.

CHAPTER 3

THE DESIGN OF A TERRAIN MODEL

3.1 Introduction

The original quad-tree representation is constructed as a simple data structure containing pointers to the four sub-trees and a further pointer to the information relating to the node as described in chapter 2. An alternative representation is the linear quad-tree which avoids the storage overhead of pointers by encoding the level of subdivision of each node, and storing the tree as a linear list of nodes. This representation of information can exploit the physical structure of specific data and thus reduce the number of access operations to retrieve data from the tree. Generally, a linear quad-tree has the following characteristics [Garg82a]:

- Only 'black' nodes are stored - storage and efficiency depend only on the number of black nodes.
- The encoding method used for each node incorporates adjacency properties relating to the four principal 'directions' of a quad-tree.
- The node representation implicitly encodes the path from the root to the node, pointers are totally eliminated.
- The linear nodes list is ordered in a linear sequence which is advantageous in searching trees and in file organisation.

Most quad-tree representations are applied to a 2-D compression of a binary image. However, in a GIS system, a geographic area usually contains multiple features such as elevation, land-use and topographic data. Thus, for applications which require the reorganisation of multiple features of terrain area, either a collection of binary quad-trees is used, with one quad-tree for each feature value [Meno87, Fabb86], or locational codes are used as storage addresses for these features [Cebr85, Chen86, Mark86, Shaf89].

In airborne applications, if a multi-layer of quad-trees is implemented to represent terrain elevation data, it is necessary to access a set of files to perform operations such as neighbour finding, connectivity, or adjacency processing rather than a single quad-tree.

Since terrain elevations are three-dimensional points in the terrain space, it is straightforward to convert the continuous surface-like terrain to oct-tree representations. However, the standard oct-tree encoding method does not achieve storage efficiency, and accordingly, oct-trees have not been widely adopted to represent terrain. If the terrain is treated as an object and encoded according to the criterion for standard oct-trees, then all the voxels between the bottom and uppermost surface will be encoded into the nodes of an oct-tree, and potentially, a large number of redundant nodes will be generated. Even if the nodes are merged as a result of a common attribute, the nature of continuous terrain surface may generate a large number of nodes. The nodes beneath the surface nodes are meaningless in the context of terrain applications. The only relevant nodes in an oct-tree are the surface nodes (or voxels). As the redundant nodes below the surface consume unnecessary memory space, only the surface of the 'terrain object' or the peak points need to be encoded into oct-trees codes.

The approach of using locational codes as storage indices offers an efficient method to reorganize terrain data if the number of nodes in the tree can be reduced. As terrains are strictly single-valued continuous surfaces, the size of an oct-tree can be reduced by discarding parts of the oct-tree below the terrain surface. This is accomplished by a projection of the terrain surface.

Independent of the elevation of a peak point, after projection, each element in the terrain matrix defines a facet of the 'terrain object'. This is a 2^n x 2^n bottom surface with respect to the corresponding three-dimension object. The locational codes of this bottom facet of a 2^n x 2^n x 2^n object in three-dimensional space, have the same representations as a 2^n x 2^n region in two-dimensional space represented by the terminal nodes of complete quad-trees, where the resolution parameter 'n' indicates the degree

of refinement of the image and the object. Each leaf node of the bottom facet quad-trees corresponds to one, and only one, element in the terrain elevation matrix. Thus the locational code of the projection plane can be used as the index of a terrain oct-tree.

3.2 Basic schema

In the design of terrain models, the methods used to represent image data by means of linear quad-tree and oct-tree are applied to DTED. These transformations of DTED follow directly from the Morton encoding sequence and will be described in the following sections. Before discussing the terrain modelling method, the following conventions and encoding methods used in linear quad-trees and oct-trees are adopted:

3.2.1 Subdivision

In the quad-tree (planar) case, the NW quadrant is encoded as 0, the NE as 1, the SW as 2, and the SE as 3. Each black pixel is then encoded as a weighted quaternary code (digits 0, 1, 2, 3) in base 4, where each successive digit represents the quadrant subdivision from which it originates. Thus, the digit of weight 4^{n-h}, $1 \leq h \leq n$, identifies the quadrant to which a pixel belongs at the h-th subdivision.

In the oct-tree (space) case, in order to support the terrain applications, an octant is divided into 8 cubes and the 'Top' and 'Bottom' (T and B) notation represents the four uppermost cubes and the four lowest cubes respectively. We then encode octant NWB with 0, octant NEB as 1, octant SWB as 2, octant SEB as 3, octant NWT as 4, octant NET as 5, octant SWT as 6, octant SET as 7. The origin (0,0,0) is located at the corner of octant NWB. The co-ordinate system and the octant notation which were adopted are illustrated in Figure 3.1.

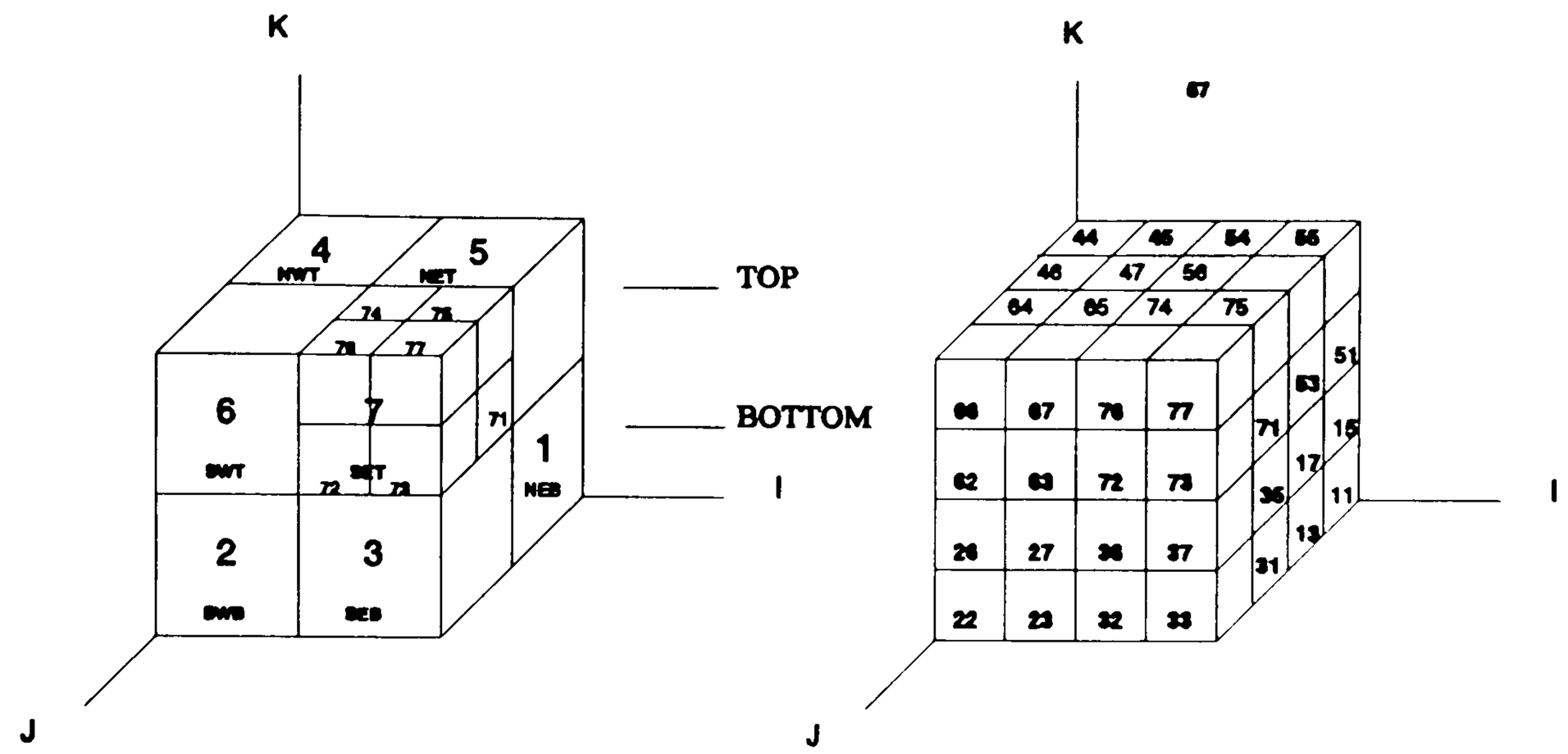


Figure 3.1 The co-ordinate and notations of terrain oct-trees.

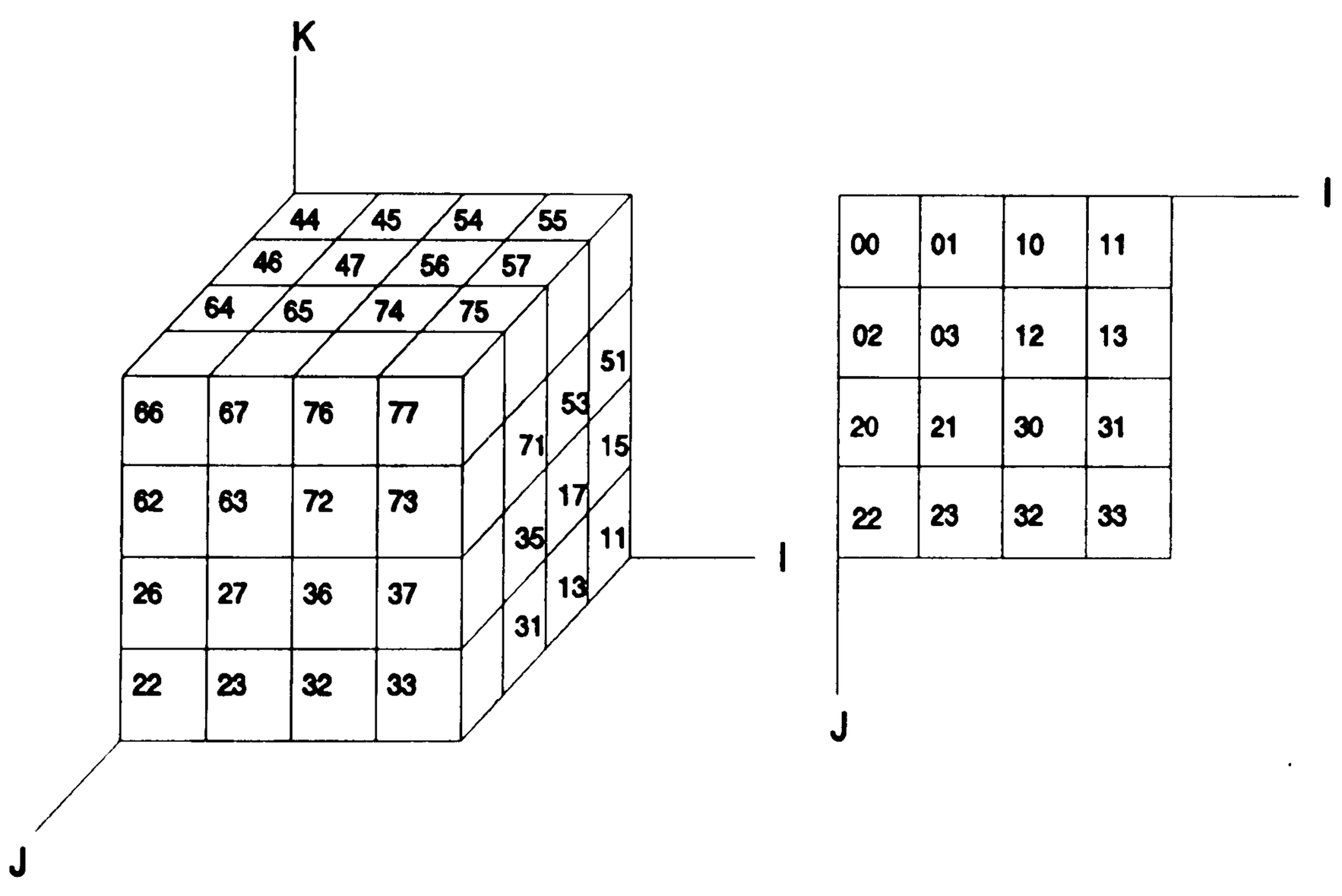


Figure 3.2 An oct-tree representation of a 4x4x4 cubic and its corresponding IJ plane projection codes (lowest layer of oct-trees).

3.2.2 Units

The lowest level of an oct-tree is equal to the quad-tree representing the region which covers the terrain, as shown in Figure 3.2. Similar to the *pixel* in two-dimensional case, the cubic pixel or *voxel* is the element of a $2^n x 2^n x 2^n$ three-dimensional space. The length of a cube in the *I, J, K* directions defines a single voxel unit. The *size* of a $2^d x 2^d x 2^d$ octant is defined to be 2^d. The *level* of the node corresponding to the octant is defined to be d, therefore, the node corresponding to a unit cube is at level 0 (the bottom level) and the root of the oct-tree is at level *n*.

3.2.3 Projection Planes

The pair of integers (*I,J*) with *I, J* = 0,1,..,2^n-1 identifies the position of an elevation in the 2^n x 2^n array. The triplet (*I,J,K*) describes the position of an elevation point (the same size as a voxel) of terrain in three-dimensional space, with *I, J, K* = 0,1,2,...,2^n-1, where (*I, J*) match the (column,row) values of a 2^n x 2^n input elevation matrix. The IJ plane is also defined as the *projection plane* of an oct-tree terrain (as shown in Figure 3.2); the lines of projection are parallel to the positive K axis. The value n is a resolution parameter indicating the degree of refinement of the terrain.

3.2.4 Encoding Arithmetic

The linear quad-tree encoding procedure consists of mapping (*I,J*) into a 2-D locational code [Garg82a], that is an integer Q in base 4, which expresses the successive quadrants to which the (*I,J*) pixel belongs. Q can be described by the following expansion:

$$Q = q_{n-1}\, 4^{n-1} + \ldots + q_1 4^1 + q_0 4^0, \qquad 3.1$$

where

$$q_l = d_l 2^1 + c_l 2^0, \quad l = 0, 1, \ldots\ldots, n-1. \qquad 3.2$$

To find Q, I and J are given as follows:

$$I = c_{n-1}\,2^{n-1} + c_{n-2}\,2^{n-2} + \ldots + c_0. \qquad 3.3$$

$$J = d_{n-1}\,2^{n-1} + d_{n-2}\,2^{n-2} + \ldots + d_0.$$

Thus, given a pixel with column-index I and row-index J, encoding can be carried out by expressing I and J in their binary form, multiplying J by 2 (to base four) and adding this product to I giving Q. For example, if $I = 15$ and $J = 9$, Q is given by $Q=(J*2+I)_4 = (2002+1111)_4 = 3113$. This encoding reflects the quad-tree convention as defined above.

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0002	0003	0012	0013	0102	0103	0112	0113	1002	1003	1012	1013	1102	1103	1112	1113
0020	0021	0030	0031	0120	0121	0130	0131	1020	1021	1030	1031	1120	1121	1130	1131
0022	0023	0032	0033	0122	0123	0132	0133	1022	1023	1032	1033	1122	1123	1132	1133
0200	0201	0210	0211	0300	0301	0310	0311	1200	1201	1210	1211	1300	1301	1310	1311
0202	0203	0212	0213	0302	0303	0312	0313	1202	1203	1212	1213	1302	1303	1312	1313
0220	0221	0230	0231	0320	0321	0330	0331	1220	1221	1230	1231	1320	1321	1330	1331
0222	0223	0232	0233	0322	0323	0332	0333	1222	1223	1232	1233	1322	1323	1332	1333
2000	2001	2010	2011	2100	2101	2110	2111	3000	3001	3010	3011	3100	3101	3110	3111
2002	2003	2012	2013	2102	2013	2112	2113	3002	3003	3012	3013	3102	3103	3112	3113
2020	2021	2030	2031	2120	2121	2130	2131	3020	3021	3030	3031	3120	3121	3130	3131
2022	2023	2032	2033	2122	2123	2132	2133	3022	3023	3032	3033	3122	3123	3132	3133
2200	2201	2210	2211	2300	2301	2310	2311	3200	3201	3210	3211	3300	3301	3310	3311
2202	2203	2212	2213	2302	2303	2312	2313	3202	3203	3212	3213	3302	3303	3312	3313
2220	2221	2230	2231	2320	2321	2330	2331	3220	3221	3230	3231	3320	3321	3330	3331
2222	2223	2232	2233	2322	2323	2332	2333	3222	3223	3232	3233	3322	3323	3332	3333

Table 3.1. Quaternary codes for n=4.

Merged nodes are denoted by an integer > 3 and each digit in the code can then be represented by a 3 bit field in a record. Decoding is the converse process; for example, if Q = 2131, from equations 3.1 and 3.2, $I = 0111_2 = 7$. From above, $2J = (Q\text{-}I)_4$, thus $2J = (2131\text{-}0111)_4 = 2020$, or $J = 1010_2 = 10$. In the case of a merged node (a code with X or 4) only the first pixel needs to be decoded. This is obtained by

equating all 4s to 0, since it is known that there are as many consecutive values of I and J as 2^m; where m is the number of 4s in the node. For example, if Q = 2044, equating Q to 2000, $I = 0000_2 = 0$ and $J=1000_2=8$. All pixels corresponding to I = 0,1,2,3 and J = 8,9,10,11 are known to be present. This method of encoding requires $O(n*B)$ time [Garg83] where n is the resolution parameter and B is the number of black pixels. Table 3.1 illustrates the Quaternary codes for n=4.

For the three dimensional case, the encoding of a voxel Q is a mapping of its spatial position (I,J,K) into its corresponding octal code and is a direct extension of the quad-tree method. In Gargantini's [Garg82c] linear oct-tree encoding method, a voxel Q is represented by an octal integer in a weighted system where the digit of weight 8^{n-1} identifies the largest octant according to the encoding method introduced in the previous section. The digit of weight 8^{n-2} identifies the second largest octant and so on. The voxel Q is described by the following expansion:

$$Q = q_{n-1}8^{n-1} + q_{n-2}8^{n-2} + \ldots + q_0 8^0, \qquad 3.4$$

where q_l, $l=0,1,\ldots,n-1$, is one of the set of digits $\{0,1,2,3,4,5,6,7\}$.

The encoding scheme is given as follows:

First, the binary representation of I, J, K, (the digit in c_l,d_l,e_l) is formed as follows:

$$\begin{aligned} I &= c_{n-1}2^{n-1}+\ldots+ c_l2^l +\ldots+ c_02^0, \\ J &= d_{n-1}2^{n-1}+\ldots+ d_l2^l +\ldots+ d_02^0, \\ K &= e_{n-1}2^{n-1}+\ldots+ e_l2^l +\ldots+ e_02^0. \end{aligned} \qquad 3.5$$

Then, the coefficients q in $Q=q_{n-1}8^{n-1} + q_{n-2}8^{n-2} + \ldots + q_0 8^0$, are determined by

$$q_l = e_l2^2 + d_l2^1 + c_l2^0, \quad l=n-1,n-2,\ldots,0. \qquad 3.6$$

Figure 3.3 illustrates an example of the linear oct-tree encoding scheme. In the figure, an object is encoded into a set of octal codes according to the location of the voxel elements within the object. This object is therefore represented by the set of nodes

$$<10,11,12,13,14,15,16,17,31,33,35>.$$

If eight voxels belong to the same octant, they can be grouped together by replacing the digits relative to the common octant by a number, which is denoted by 8. The use of the number 8 enables the octal code to be sorted in an ascending sequence and merged to upper levels of the data structure. The final representation of the three-dimensional object in Figure 3.3 is then given by $<18,31,33,35>_8$. The structure used here is a simple integer array. For the position I=3, J=2, and K=1 in Figure 3.3, $q_1=3$ and $q_0=5$ is obtained; therefore, $<35>_8$ identifies the voxel with subscripts (I,J,K).

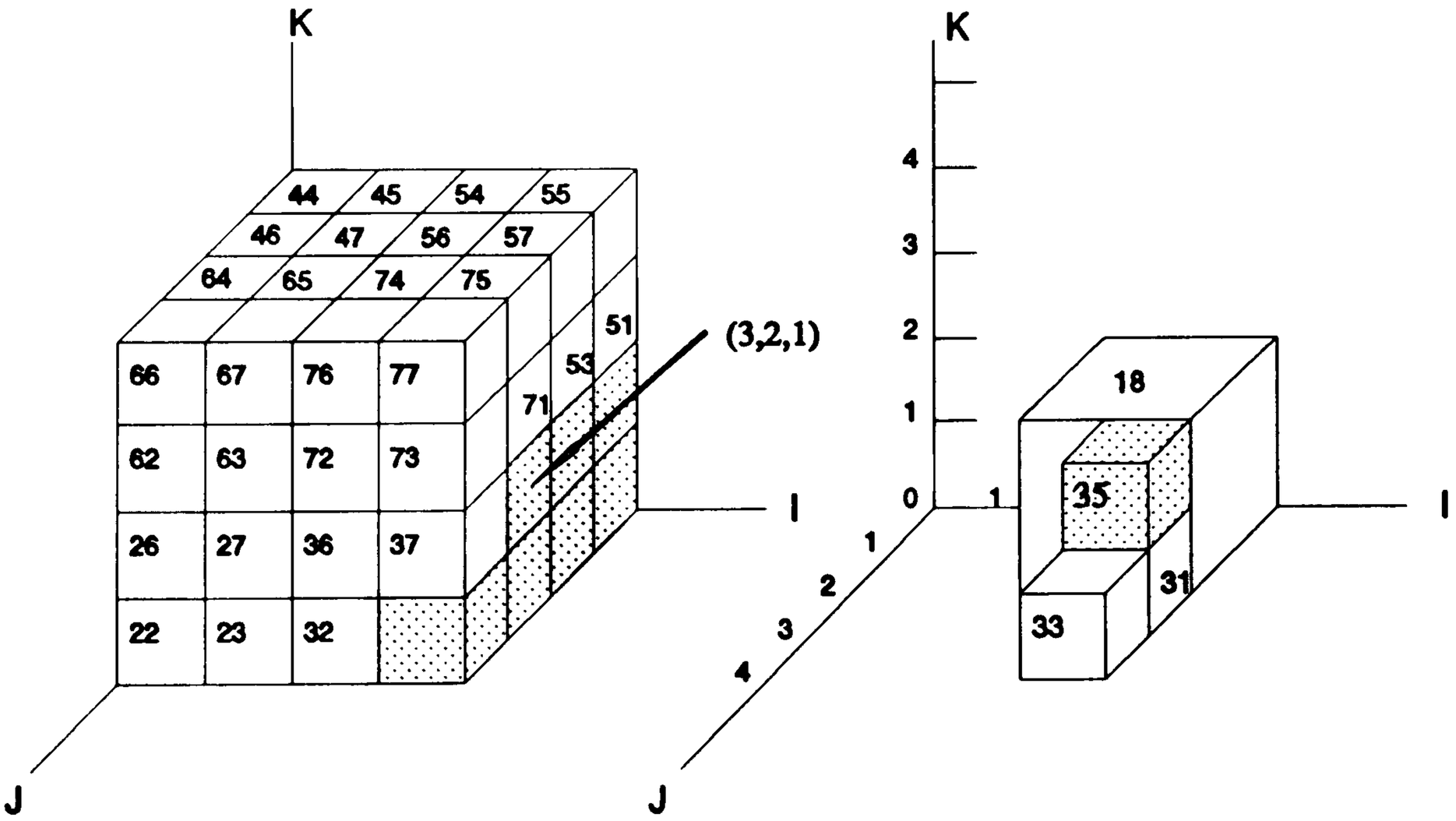


Figure 3.3 an example of the linear oct-tree encoding scheme.

In an oct-tree representation, the locational codes in 3-D space are shown represented by base-8 digits (or octal values), that is each (I,J,K) triplet corresponds to a group of three octal digits (i.e. the (I,J,K) bit triplet corresponds to a single base 8-digit). However, as the digit 8 represents merged nodes and also indicates the node size, four bits are used to represents the digits set {0,1,2,3,4,5,6,7,8}.

For example, the code <358> denotes that the voxel belongs to the SE-BOTTOM (3) octant in the first subdivision and the NE-TOP (5) octant in the second subdivision. A digit 8 indicates that a merge has occurred in the third subdivision. This approach uses a fixed number of digits to represents locational codes. Decoding is the converse process, similar to the quad-tree case.

The octal code of projection on the IJ plane is obtained by ignoring the contribution of e_1 to each of the Q digits. This is readily achieved by taking each q_l modulo-4. For instance, the projections of $<545, 735, 627>_8$ are respectively, $<101, 331, 223>_8$ as shown in Figure 3.4. In the case of nodes which are merged to an upper

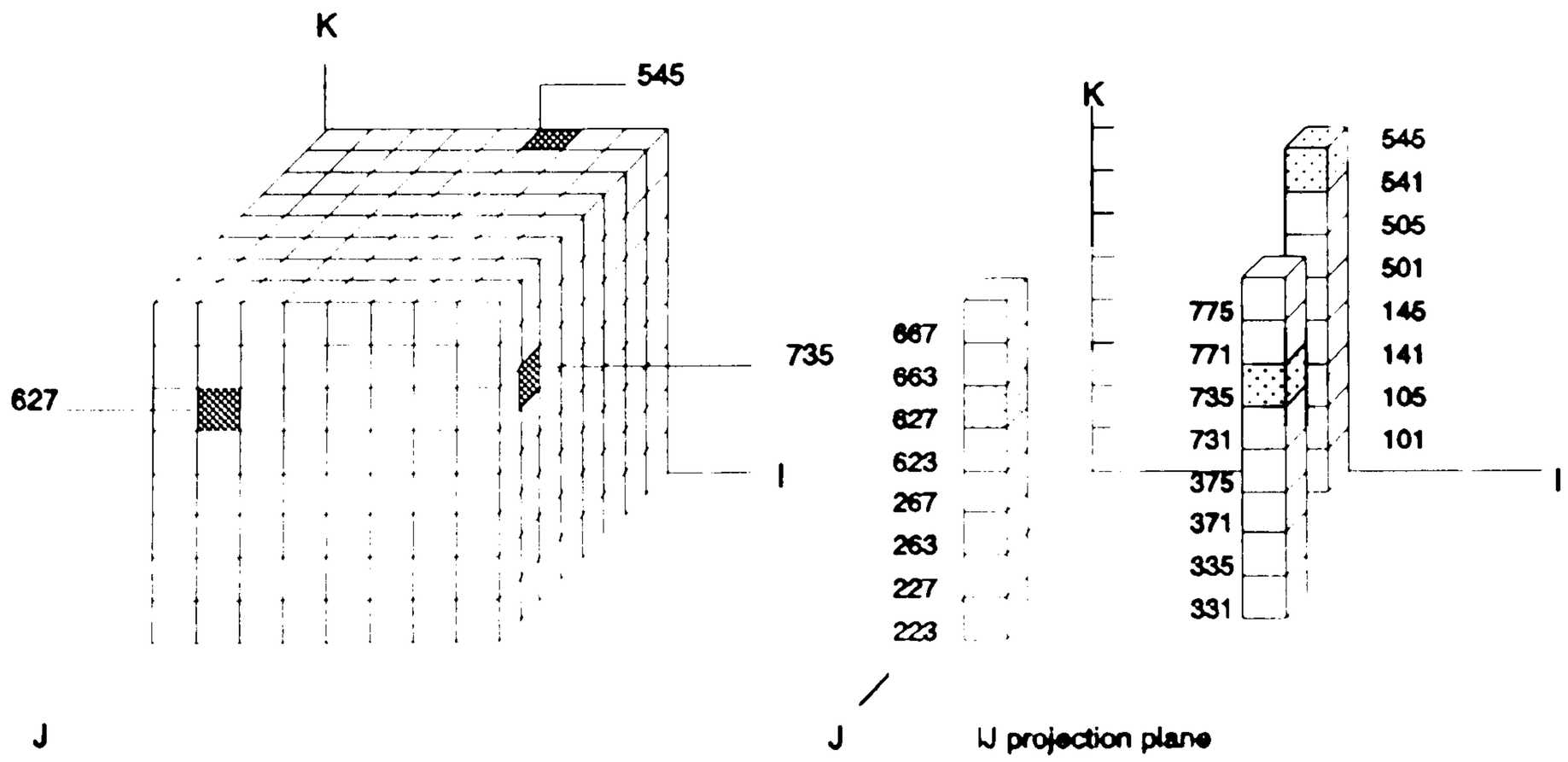


Figure 3.4 An example of oct-tree nodes and their projection codes.

level node, where a digit of q_l equals 8, the digit 8 is kept unchanged, and other digits are encoded as described. $<358>_8$ for instance, has a projection code on the IJ plane of $<318_k>_8$, where 8_k denotes the set of digits {0,1,2,3}.

In the following sections, a terrain oct-tree model is proposed based on the linear quad-tree construction method adapted for a 3-D locational code. This terrain model is a structure with both quad-tree and oct-tree features. It represents a 3-D terrain surface but enables operations on the terrain to be performed in 2-D space.

3.3 Terrain Oct-tree Design

3.3.1 Objectives

Each unit cube of an object in three dimensional space can be represented in a binary format. It is straightforward to build an oct-tree from the corresponding 3-D binary array. However, the standard linear oct-tree encoding method described in section 3.1 is not suitable. In this section, a modified version of the locational code based oct-tree terrain model is proposed. The following topics related to terrain oct-tree encoding will be discussed: how to specify a block in space, how to specify a leaf node in a terrain oct-tree, and how to convert the spatial co-ordinates of a block to the locational code of its corresponding node.

A terrain oct-tree model is organized as a sorted list of leaf nodes, where each node is represented as a single integer and represents a homogeneous set of elements of the grid file. Unlike other quad-tree methods which are used to represent terrain, the elevation data is either defined as an attribute appended to its locational codes [Fabb86, Meno87] or as a file in which Morton numbering of a cell provides the index of the storage location for the elevation [Cebr85, Chen86].

In order to match the upward direction of a terrain surface (refer to section 3.2.1), the co-ordinate system and octants are made so that the downward projection

of the terrain surface has the same locational codes as a 2^n x 2^n 'complete' quad-tree. This is illustrated by the examples in Figure 3.1 and 3.2. Every element in a 2-D array has a corresponding leaf node in its quad-tree representation. For the convenience of this description, the locational codes of an oct-tree are termed "3-D locational codes"; where the locational codes of projected quad-trees are obtained from a terrain oct-tree, they are termed "2-D locational codes".

The encoding of a terrain array into a terrain oct-tree is not limited to the mapping of its array co-ordinate (*I,J*) into the corresponding locational codes, it also includes the mapping of scaled elevation values and related homogeneous information into the corresponding locational codes [Alle92]. This homogeneity can be obtained by applying a vertical *scaling factor* to each elevation in the grid file so that elevations in the same projection quadrant have a common scaled value. This *scaling factor* which plays an important role in the oct-tree terrain model dominates the number of nodes of the terrain oct-tree (described in chapter 6). The scaling factor is defined as an increment, relative to an elevation range from the base plane to the terrain surface.

The continuous change of terrain topography is the main reason for the introduction of a scaling factor which divides the elevation into bands. In the standard oct-tree method, the equal unit length of a cube in the *I, J, K* directions defines a voxel unit. In the oct-tree terrain model, the length of a cube in the K axis direction depends on the scaling factor of the elevation. Non-linear scaling can be applied to the terrain elevation data in order to maximise the distribution of elevation data and to minimise the number of bands, depending on the accuracy of the application. The experimental results of applying different scaling factors to the terrain elevation data are presented in chapter 6.

This method of mapping a peak point (*I,J,K*) into a locational code of a terrain oct-tree is similar to the method used by Gargantini [Garg82c]. However, there are three major differences:

- Firstly, the scheme of merging or grouping homogeneous nodes is redefined.

- Secondly, the use of an octal integer in a weighted system to represent the subdivision of octants (described in expansions 3-4 to 3-6) is replaced by an equivalent hexadecimal weighted system.
- Thirdly, a fourth parameter S is added to represent the size of a leaf node.

A single 3-D locational code can represent an area of terrain surface by its three dimensional co-ordinates and size information. Unlike the Gargantini's method [Garg82c], the size information is obtained by counting the appearance of a digit '8' or 'X' (don't care value).

3.3.2 Encoding and Decoding

In defining the terrain oct-tree encoding scheme, the voxel Q can be described by the following expansion:

$$Q = q_{n-1}16^{n-1} + q_{n-2}16^{n-2} + \ldots + q_0 16^0, \qquad 3.7$$

where q_l is one of the set of digits {0,1,2,3,4,5,6,7,8,c} and $l = 0,1,\ldots,n\text{-}1$ denotes the depth of the node. The binary representation of *I*, *J*, *K*, *S* (the digit in c_l,d_l,e_l,f_l) is similar to the scheme adopted for the linear oct-tree and is formed as follows:

$$\begin{aligned} I &= c_{n-1}2^{n-1}+\ldots+ c_l2^l +\ldots+ c_02^0, \\ J &= d_{n-1}2^{n-1}+\ldots+ d_l2^l +\ldots+ d_02^0, \\ K &= e_{n-1}2^{n-1}+\ldots+ e_l2^l +\ldots+ e_02^0, \\ S &= f_{n-1}2^{n-1}+\ldots+ f_l2^l +\ldots+ f_02^0. \end{aligned} \qquad 3.8$$

Then the coefficients q in $Q = q_{n-1}16^{n-1}+q_{n-2}16^{n-2}+\ldots +q_016^0$, are determined by

$$q_l = f_l2^3 + e_l2^2 + d_l2^1 + c_l2^0, \quad l = n\text{-}1,n\text{-}2,\ldots,0. \qquad 3.9$$

Since $q_l = f_l*8+e_l*4+d_l*2+c_l$, each q_l is always $<=15$ and there is no carry propagation in the evaluation of Q which implies that simple logical operations are adequate for the encoding and decoding of terrain oct-trees.

The following formulas describe the decoding process from the hexadecimal code to the (I,J,K) and S form. The operation & is defined as the bit-wise AND operator. *I*, *J*, *K* and *S* are derived as follows:

$$I = \sum_{l=1}^{n-1} (q_l \,\&\, 1) \times 2^l \qquad 3.10$$

$$J = \sum_{l=1}^{n-1} (q_l \,\&\, 2) \times 2^l \qquad 3.11$$

$$K = \sum_{l=1}^{n-1} (q_l \,\&\, 4) \times 2^l \qquad 3.12$$

$$S = \sum_{l=1}^{n-1} (q_l \,\&\, 8) \times 2^l \qquad 3.13$$

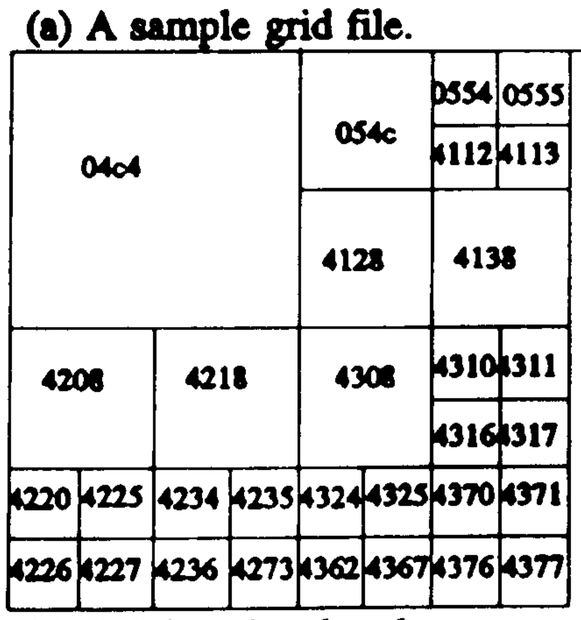
Figure 3.5 gives an example of the application of a scaling factor to a terrain array. In this example, each index pair (*I,J*) and its respective scaled elevation *K* is encoded into 3-D locational codes of the corresponding spatial point (*I,J,K*), where a 'ceiling function' is applied to each elevation element to obtain the scaled value *K*. The

320	320	320	320	330	340	340	340
320	320	330	330	340	340	350	360
330	330	340	340	350	360	360	370
340	340	340	340	350	360	370	380
350	350	360	360	360	370	380	380
370	380	380	390	380	390	400	420
390	400	420	430	430	440	450	450
400	420	440	460	480	500	520	540

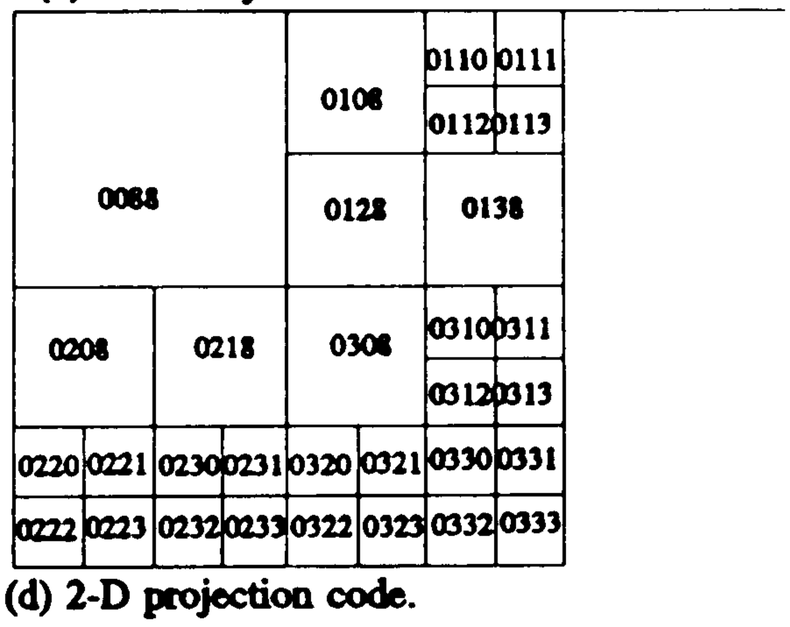
(a) A sample grid file.

7	7	7	7	7	7	7	7
7	7	7	7	7	7	8	8
7	7	7	7	8	8	8	8
7	7	7	7	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	9	9
8	9	9	9	9	9	10	10
9	9	9	10	10	11	11	11

(b) Scaled by 50.



(c) 3-D locational code.



(d) 2-D projection code.

Figure 3.5 A DTED encoded terrain oct-tree with 50 meter scaling factor.

use of a ceiling function is an approximation process. Consequently, the encoded terrain oct-tree nodes have 'rounded' elevations which are higher than the original terrain array.

3.3.3 Formatting and Characterization

The encoding and decoding methods are described as follows:

In oct-tree encoding, an octant is encoded with a locational code which is one of the set of digits {0,1,2,3,4,5,6,7}. A node in this oct-tree is therefore specified by the path, given by the locational codes, originating from the root of the tree to the node. If eight voxels belong to the same octant, they are grouped together by replacing the digits of the common octant by a 'flag' bit denoted by 8 (see Figure 3.3 in section 3.2).

Usually, 4 bits are necessary to represent the digit of a locational code. Although an alternative method is feasible without using a flag bit to label the size of a node, an explicit field still has to be appended to the locational codes. Since a 4 bit digit can represents integers with a range from 0 to 15, some of the hexadecimal codes are unused in a linear oct-tree. The fourth bit of each digit can be used independently to represent the size S (refer to section 3.2.2) of a node without interfering with the *I*, *J*, *K* value.

According to the definition of oct-trees given above, the grouping of a sub-octant exhibits the feature that only odd values occur along the K axis. That is, the upper layer of an octant has underlying sub-octants {4,5,6,7}. The scaled elevation K is given by 2^n-1 where the resolution parameter n also determines the resolution in the IJ projection plane, equal to 2^n. The degree of grouping is limited when encoding continuously changing terrain surface because almost all leaves will be at level 0.

By exploiting the relationship that the projection of a terrain surface generates a complete quad-tree representation, the grouping process can be performed in quadrants instead of octants. In other words, the terrain surface voxels can be merged

and marked if the IJ plane projection of four blocks falls into the same quadrant of the corresponding quad-tree with an equal *K* value.

Under the scheme discussed above, a block of terrain surface with equal *K* values can be uniquely identified by the size of the block and the co-ordinates of its NW corner. Assuming a 2-D input DTED of side length 2^n, the locational code of each leaf node of side length 2^k consists of n digits where the k trailing digits contain an '8' or 'C', and the leading n-k digits contain the locational codes defining the path from the root of the tree. Note that the merging of a leaf node is based on the quadrants of the projection plane, the locational code along the path should be represented by 0, 1, 2, and 3. However, the third and forth bits of each digit are reserved for the scaled value *K* and the size value S respectively. After interleaving *(I,J,K)* and *S*, the locational code is one of the set of digits {0,1,2,3,4,5,6,7,8,C}. The times of digit '8' or 'C' appears in the location code and gives the size information of a node.

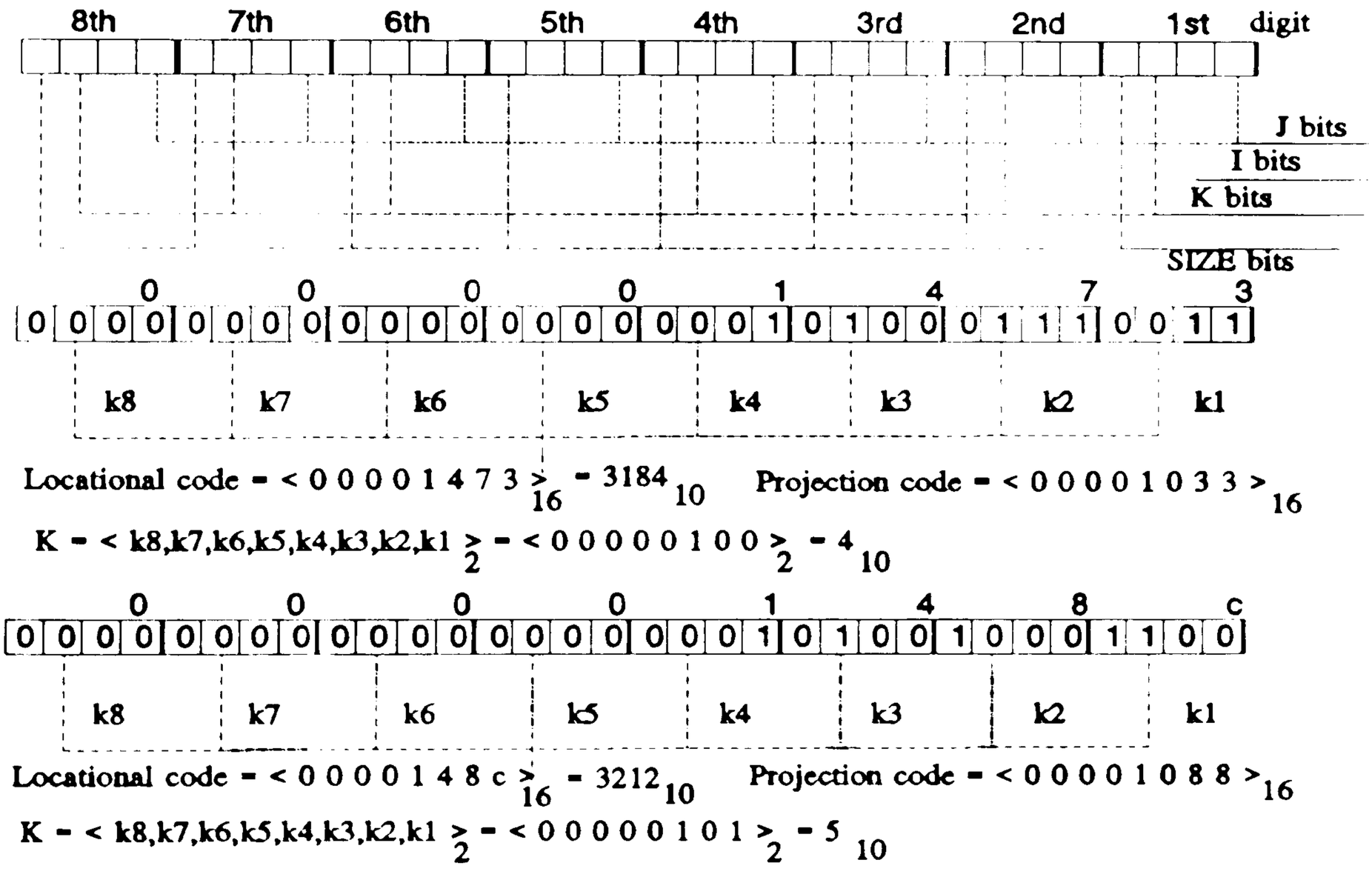


Figure 3.6 Data format of the node of a terrain oct-tree.

As shown in the expansions 3.7 to 3.9, the locational code of a terrain oct-tree node is formed by bit interleaving the triplet (I,J,K) and S as an hexadecimal value such that the S bit precedes the K bit, the K bit precedes the J bit, and the J bit precedes the I bit at each position. For example, in figure 3.6, the combination of the 4^{n-1}th bits (i.e., the combination of the first bit of each digit) of a node Q is equal to the binary representation of value I; this is repeated for the $(4^{n-1}+1)$th bits to J, the $(4^{n-1}+2)$th bits to K and the $(4^{n-1}+3)$th bits to form the size value S respectively, where n is the resolution parameter. Although the four components of a node (i.e., I, J, K, S) are totally independent, they are represented by a single value. Both the encoding and decoding process can be performed by the use of bit-wise logical operations under this bit interleaving scheme.

As the terrain array is a 2-D array, the corresponding IJ projection quad-tree is a one-to-one mapping of nodes from the terrain oct-tree to a quad-tree. Usually, the 2-D locational codes are treated as the index value of a quad-tree and the locational codes can also be applied as the index value of a terrain oct-tree.

The addressing schemes used in terrain oct-trees are the Morton numbering sequence, which are formed by bit interleaving as previously described. The 2-D locational code can be obtained by the use of modulo-4 operations on the I, J bits of the 3-D locational code, where the K bits are assigned as zero, and the S bits are left unchanged. Assuming a 3-D locational code, Q_{3d} is represented by the form given in equations 3.7-3.9 and its 2-D locational code Q_{2d} can be expressed as follows:

$$Q_{2d} = q_{n-1}16^{n-1} + q_{n-2}16^{n-2} + \ldots + q_0 16^0, \qquad 3.14$$

where

$$q_l = f_l 2^3 + \mathrm{mod}(d_l 2^1) + \mathrm{mod}(c_l 2^0), \quad l=n-1,n-2,\ldots,0. \qquad 3.15$$

For example, in Figure 3.6, the projection of a node Q $<00001473>_{16}$ on the IJ plane is $<00001033>_{16}$; for the merged node Q $<0000148c>_{16}$ several digits are greater than 8 and the projection code is $<00001088>_{16}$. Table 1 in section 3.2 also

serves as a projection table as well as an index of the elements of a terrain matrix. Note that the digits are now in a hexadecimal representation.

The identity features between an oct-tree and its corresponding 2-D locational codes allow most of the operations on terrain data (adjacency searching, neighbour-finding and connectivity) to be performed in two-dimensional space. In order to cover a wide range of applications in terrain navigation, it is necessary to combine two-dimensional and three-dimensional representations of terrain in specific algorithms. Figure 3.2 shows the relationship between 2-D, 3-D locational codes of a terrain oct-tree. The systematic approach described for the conversion between locational codes and spatial co-ordinates of terrain surface points greatly facilitates operations performed in the construction and manipulation of terrain models.

3.4 Encoding and Decoding Algorithms

In this section, the formal descriptions of encoding, decoding and projection algorithm are described. Since individual bits within a number can be accessed in the C programming language, conversions between co-ordinates and locational codes are straightforward.

The algorithm for determining the terrain oct-tree locational code is given in pseudo-code* below. In the algorithm, the input parameters are *I, J, K, S* where *(I,J)* are the coordinates of the NW corner, *K* is the scaled elevation and *S* is the size value of an homogeneous area. The procedure ENCODING(*I, J, K, S)* returns the corresponding locational code of the node, where n is the resolution parameter with respect to a 2^n x 2^n array. Assuming n = 8 in this algorithm. The pseudo-code LShift() and RShift() represents the bit-wise logical left and right shift operations respectively.

---

*In this thesis, the algorithms are more easily represented by pseudo-code than the more traditional flow charts.

```
procedure ENCODING(I,J,K,S);
begin
  value integer n, D, BitI, BitJ, BitK, BitS;
  value integer Mask, Node, DigitI, DigitJ, DigitK, DigitS;
  n ← 0;
  Mask ← 1;
  for each digit of the I,J,K,S values, (n<8) do
    begin
      BitI ← ( I & Mask ) ? 1:0;
      BitJ ← ( J & Mask ) ? 1:0;
      BitK ← ( K & Mask ) ? 1:0;
      BitS ← ( S & Mask ) ? 1:0;
      D ← LShift(1,LShift(n,2));
      increase each component value according to its bits position;
      DigitI ← ( DigitI + LShift(BitI,D  ));
      DigitJ ← ( DigitJ + LShift(BitJ,D+1));
      DigitK ← ( DigitK + LShift(BitK,D+2));
      DigitS ← ( DigitS + LShift(BitS,D+3));
      Mask ← LShift(Mask,1);
      n ← n + 1;
    end;
  Node ← ( DigitI | DigitJ | DigitK | DigitS );
  return(Node);
end;
```

The complementary procedure DECODING for deriving the planar co-ordinates, scaled elevation and size information of a leaf node from its locational code is given in pseudo-code below. In the algorithm, given the locational code of a node, the procedure returns *(I, J, K, S)* where *I, J* is the (Column, Row) index of the element in the NW corner, *K* is the scaled elevation and *S* is the size of the corresponding homogeneous area of the terrain. Assuming a 32 bit integer is used to represents the locational code, a 2^8 x 2^8 matrix can be represented by a single integer.

```
procedure DECODING(int Node);
begin
  value integer n, BitI, BitJ, BitK, BitS;
  value integer MaskI=1, MaskJ=2, MaskK=4, MaskS=8, Node;
  value integer I=0, J=0, K=0, S=0;
  n ← 0;
  for each digits of a node, n<8 do
    begin  retrieve each component of a node bit by bit;
      BitI ← ( Node & MaskI ) ? 1:0;
      BitJ ← ( Node & MaskJ ) ? 1:0;
```

```
BitK ← ( Node & MaskK ) ? 1:0;
BitS ← ( Node & MaskS ) ? 1:0;
if (BitS != 1), not a merged node  then
  begin
    I ← I + LShift(BitI,LShift(1,n));
    J ← J + LShift(BitJ,LShift(1,n));
  end;
  K ← K + LShift(BitK,LShift(1,n));
  S ← S + BitS;
  n ← n + 1;
end;
```

The algorithm for obtaining the projection code of a node is as follows. Given a 3-D locational code of a node, the procedure PROJECTION derives the IJ plane projection code. The projection code is obtained by a bit-wise logical **and** operation between the Node and 'Mask' value.

```
procedure PROJECTION(int Node)
begin
  value integer n, Count, BitS;
  value integer Projection, Mask, MaskS, Node;
  n ← 0;
  Count ← 0;
  MaskS ← 8;
  BitS ← 1;
  for each digit of a node (n < 8) with size bit = 1 do
    begin determine the size of a node
      BitS ← ( Node & MaskS ) ? 1:0;
      if (BitS != 0)
        begin  MaskS shif to left
          MaskS ← LShift(MaskS,4);
          Count ← Count+1;
          n ← n + 1;
        end;
    end;
  generate the mask pattern of 2-D locational code.
  Mask ← ( LShift(33333333₁₆,Count) | RShift(88888888₁₆,8-Count) );
  Projection ← ( Node & Mask ); reset S bits to 0
  return(Projection);
end;
```

The encoding and decoding processes pass through their main loop to perform a constant amount of bit-wise operations for each digit of a locational code. The time to execute both algorithms will then be determined by the number of passes through the

loop, plus a constant time for initialization. Both the encoding and decoding algorithms require time which is proportional to the resolution parameter n for a 2^n x 2^n input array and the time performance is $O(n)$. The cost of obtaining the projection code also requires time proportional to n.

3.5 Constructing Terrain Oct-trees

Gargantini's linear quad-tree construction algorithm [Garg82a] examines an input binary array in row order. After all the black pixels have been encoded into their corresponding quaternary codes, the locational codes are sorted and stored in an array or list. The pixels are then compressed by checking if any four pixels have the same representation (ignoring the last digit); if four pixels match the condition, then they are eliminated from the list and replaced with a code of (n-1) quaternary digits including a tag given by the number 4. Although the approach discussed above is simple, it is wasteful of storage since many temporary nodes are created. Moreover, the available memory may be insufficient for the resultant quad-tree [Same90a]. An alternative algorithm for converting a raster image to a linear quad-tree is to insert each pixel of an image into the quad-tree in scan-line order. Pixels making up larger nodes will be merged together by the quad-tree insert routine. This algorithm executes in time proportional to the number of pixels in the image [Same81]. Moreover, additional searching is needed to locate neighbours, as well as the four siblings that are candidates for the merging process.

Lauzon's *2D Run Encoding* method [Lauz85] constructs a linear quadtree based on Morton numbering. He observed that there will generally be a sequence of consecutive records having the same attribute value in constructing the quad-tree. Knowing that a leaf node has a specific and testable definition, it is possible to combine consecutive records efficiently (with the same attribute value) without losing any of the inherent quad-tree topologic or spatial relations of the quad-tree.

Clearly, the Run Encoding method is an efficient method for representing DTED files and has therefore been adopted for this research program. By applying this construction scheme to a DTED file, a terrain oct-tree construction algorithm is proposed. In the construction algorithm, each element of the terrain array is examined once and only once. A node is only created if it is a leaf node. In addition, merging and neighbour finding is not required. This construction is similar to the construction of a quad-tree of binary image, however the input data is multi-valued and the locational code itself is based on the format of an oct-tree.

When constructing a terrain oct-tree, each element in the underlying array is visited in a Morton numbering sequence. The encoding of a leaf node is performed by checking the equality of K values in consecutive elements in Morton sequence. Whenever the K value changes, a set of intervening leaf nodes are generated corresponding to the consecutive elements. The formulation of intervening leaf nodes contained in consecutive and equal K value elements is similar to the decoding process of *2DRE* [Lauz85]. Since any leaf node is defined by a set of $2^l x 2^l$ consecutive Morton numbers, where l is the level, and where $l = 0$ for an individual cell, it has been shown [Lauz85] that the last cell in any valid leaf node of level l, $[Morton + 1] \bmod 4^l = 0$. In other words, the value 0 suggests that a maximal leaf node may exist at a higher level, otherwise it must be a maximal leaf node.

Let equal value elements begin with Morton number M_1 and end with Morton number M_2 respectively. We first determine the sibling quadrant M_{s1} of the one corresponding to M_1. This is done by repeatedly incrementing M_1 by 4^l while the sum does not exceed M_2, and the result of $(M_1+1) \bmod 4^l$ equals 0, where l is initially 0. Once M_{s1} has been found, its block value is added to the list of nodes, M_1 is set to M_{s1} and the process is restarted. If M_2 is exceeded prior to finding a sibling block of the desired size, that is $M_{s2} = M_1+4^{l-1}$ such that $M_{s2} <= M_2$ and $M_2 < M_1+4^l$, then the block value corresponds to M_{s2}, M_1 is reset to M_{s2}, and the process is restarted. The entire process terminates when the block corresponding to M_2 has been found.

Each intervening 2^l x 2^l block is described by the Morton number of the element that occupies the upper left corner of the block and the level information l. The Morton number and the scaled value K form a triplet (I,J,K). The triplet with the size information is assigned a unique 3-D loctional code as described in section 3.3. For example, given a 2^n x 2^n terrain matrix, when a consecutive sequence of terrain elevation elements in the range 0 to 21 is detected, then according to the criteria of quadrant homogeneity, this set of elements can be grouped into a 2^2 x 2^2 block (elements 0 to 15), a 2^1 x 2^1 block (elements 16 to 19) and two single blocks representing elements 20 and 21. Figure 3.7 shows this set of elements with a K value equal to 7. The elements of a terrain array are numbered according to the order in which they are examined. Blocks having alphabetic labels are assigned according to the order in which the leaf nodes are created. Although the resulting list of terrain oct-tree nodes is not in ascending sequence, its projection codes are in increasing order. Moreover, the conversion process between Morton numbers and row and column co-ordinates is straightforward using bite-wise logical operations.

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

(a) Encoding sequence.

320	320	320	320	330	340	340	340
320	320	330	330	340	340	350	360
330	330	340	340	350	360	360	370
340	340	340	340	350	360	370	380
350	350	360	360	360	370	380	380
370	380	380	390	380	390	400	420
390	400	420	430	430	440	450	450
400	420	440	460	480	500	520	540

(b) A 8 by 8 sample elevation array

7	7	7	7	7	7	7	7
7	7	7	7	7	7	8	8
7	7	7	7	8	8	8	8
7	7	7	7	8	8	8	8
8	8	8	8	8	8	8	8
8	8	8	8	8	8	9	9
8	9	9	9	9	9	10	10
9	9	9	10	10	11	11	11

(c) Scaled by 50.

<table>
<tr><td colspan="4" rowspan="2">A
(0-15)</td><td colspan="2">B
(16-19)</td><td>C
20</td><td>D
21</td></tr>
<tr><td colspan="2">E</td><td>F</td></tr>
<tr><td colspan="4" rowspan="2"></td><td colspan="2">G</td><td colspan="2">H</td></tr>
<tr><td colspan="2">I</td><td colspan="2">J</td><td colspan="2">S</td><td>T</td><td>U</td></tr>
<tr><td colspan="4" rowspan="2"></td><td colspan="2" rowspan="2"></td><td>V</td><td>W</td></tr>
<tr><td>K</td><td>L</td><td>O</td><td>P</td><td>X</td><td>Y</td><td>b</td><td>c</td></tr>
<tr><td colspan="4"></td><td colspan="2"></td><td>d</td><td>e</td></tr>
<tr><td>M</td><td>N</td><td>Q</td><td>R</td><td>Z</td><td>a</td><td colspan="2"></td></tr>
</table>

(d) leaf nodes generated in alphabetic order.

Figure 3.7 An example of terrain oct-tree construction sequence.

In the construction of a terrain oct-tree, the procedure CONSTRUCT_TREE determines the maximal leaf nodes of a terrain oct-tree by checking the equality of scaled values K of the 2^nx2^n terrain array elements in Morton sequence. The procedure MORTON_TO_COOR is invoked to translate the Morton number into a (Col,Row) index of the terrain array. Whenever a K value is changed, the procedure MAXIMAL_NODE is invoked to generate the leaf nodes between two Morton numbers. A formal description of the terrain oct-tree construction algorithm in pseudo-code is as follows:

```
procedure CONSTRUCT_TREE(value integer Pixel[2^n][2^n])
begin
  value integer Base, Morton;
  value integer N, Col, Row, K, L, Scale, CurrentK;
  Morton, Base ← 0;
  CurrentK ← Ceiling(Pixel[0][0],Scale);
  For Each element in Morton numbering sequence do
    begin
       derive the Col and Row value by MORTON_TO_COOR(Morton);
       K ← Ceiling(Pixel[Col][Row],Scale));
       if CurrentK equals K
         Morton ← Morton + 1;/* set as current point of equal value K*/
       else
         begin
           derive the leaf nodes by MAXIMAL_NODE(Base,Morton);
           Base ← Morton; /* set as the start point of equal value K */
           Morton ← Morton + 1;/* set as current point of equal value K*/
           CurrentK ← K;
         end;
    end;
end;

procedure MORTON_TO_COOR(value integer Morton)
begin
  value integer MaskI, MaskJ, Morton;
  value integer I, J, BitI, BitJ;
  MaskI ← 1;
  MaskJ ← 2;
  I ← 0;
  J ← 0;
  for each digit of Morton value do
    begin derive the corresponding Col and Row value
      BitI ← ( Morton & MaskI ) ? 1:0;
```

```
      BitJ ← ( Morton & MaskJ ) ? 1:0;
      I ← I + BitI*(MaskI);
      J ← J + BitJ*(MaskJ);
      MaskI ← LShift(MaskI,2);
      MaskJ ← LShift(MaskJ,2);
    end;
end;
```

```
procedure MAXMIAL_NODE(value integer M1,M2)
begin
  value integer M1, M2;
  value integer K, L;
  while (M1 < M2) do derive the maximum leaf nodes between M1,M2
    begin
      L ← 0;
      while (M1+4↑L<M2) and ((M1+1)mod4↑L)=0 do L ← L+1;
      if (M1 + 4↑L > M2) then L ← L-1;
      MORTON_TO_COOR(M1);
      M1 ← M1 + 4↑L;
      ENCODING(I,J,K,L);
    end;
end;
```

Unlike the linear quad-tree, which stores only black nodes, the transformation of a DTED to a terrain oct-tree representation implies total coverage of a given terrain area. Each element belongs to its corresponding leaf node. Assuming an 2^8 x 2^8 DTED file, constructed as a terrain oct-tree, the resolution parameter n is 8 (the root is at level 8) and the 2^8 possible elevation ranges (0 - 255) are represented by 8 bits of a 32 bit locational code. Given N_L, the total number of leaf nodes, the space requirements are easily determined in terms of bits. An eight-level terrain oct-tree requires 32 bits (4 bytes) to store a node. A terrain oct-tree can therefore be stored in $32N_L$ bits of memory. In the general case, the total number of bits is $O(4*n*N_L)$.

The number of nodes generated by the above construction algorithm is dependent on the terrain source data. Clearly, a 'rough' terrain needs more storage space for the corresponding large number of nodes. One way to reduce the number of nodes is by increasing the scaling factor; the larger the given scaling factor, the fewer the leaf nodes. However, the cost of this reduction of resolution is the loss of data which is rounded to a scaled value K.

Assuming the scaled elevation K in a Morton number checking sequence changes V times, the **procedure** MAXIMAL_NODE is invoked V times. Identifying each leaf node within an interval of equal K values requires a time proportional to the resolution parameter n. If the average number of leaf nodes within an interval is A, then the average cost of invoking the procedure MAXIMAL_NODE is $O(n*A)$, and the overall cost is $O(n*A*V)$. Since $A*V$ equals the total number of leaf nodes in the tree, the complexity can be stated as $O(n*N_L)$.

3.6 Operations on a Terrain Oct-tree

3.6.1 Accessing

In order to locate the block associated with a given point in a terrain oct-tree, it is necessary to search the oct-tree structure. Searching involves the inspection of a file to extract attribute information about a specific geographic point or a set of geographic points. In a terrain oct-tree, there is a one-to-one mapping between 3-D locational codes and projection codes, and therefore, in order to locate the node containing a specific 3-D position, it is only necessary to generate the 2-D locational code. Mapping from world co-ordinates to the 2-D locational code is achieved by first applying Morton sequence encoding to the (I,J) co-ordinates (as shown in the expansions 3.7 - 3.9).

After mapping, the next process is to access a node by its 2-D locational code. Accessing is an example of the use of 2-D locational codes as indexes in terrain oct-trees. Since the nodes list of a terrain oct-tree is sorted by projection codes in ascending order, accessing a node is accomplished by applying a binary search to the list. When looking for any node in the list, the binary search performs the comparison of projection codes in the nodes list with the target 2-D locational code, although the list itself (value of the nodes) is not in sorted order.

In applying a binary search, it is necessary to check the middle element to see

if the node is contained in the element. If it is, its value is extracted and the search terminated. If the node is not found, then if the value of the node is greater than the value in the middle element, all elements below the middle element are eliminated from consideration. If the desired record is in the list, it must be above the middle element and vice versa. In any event, at most half the elements remain to be considered, and the same procedure is then applied to the remainder of the list.

The search is terminated as soon as the desired node is found or as soon as it is established that the node is not present. However, if the node is not present in the node list, it may be contained in a merged node in an upper level. This is because a merged node is represented by the locational code which is encoded from the co-ordinates of the upper left corner of the coverage block. Elements with their locational code values between two consecutive nodes are covered by the first node. The node with the projection code which is used in the last comparison loop actually covers the query node.

The purpose of accessing a node is to retrieve the elevation data and the coverage area. Once a node is located, the co-ordinates, an approximate elevation K and a size value S can also be retrieved from the node by extracting the $(4^{n-1}+2)$th and $(4^{n-1}+3)$th bits of its corresponding locational code respectively, where n is the resolution parameter. Figure 3.8 illustrates an example of retrieving a node with a 2-D locational code 0102; the key is contained in a node with a 2-D projection code 0108 which is the node 054c with scaled *K* value 7 (refer to Figure 3.5).

The following pseudo-code outlines the accessing algorithm. The procedure ACCESSING searches the N_L nodes stored in ascendent order by 2-D projection codes in the array data for the search key, where the 3-D locational codes are random. 'Found' will be true if a node is present with a projection code equal to the search key value or if the node is located in the final comparison loop of a binary search. The procedure COMPARE checks if a node contains a 2-D locational code equal to the search key value.

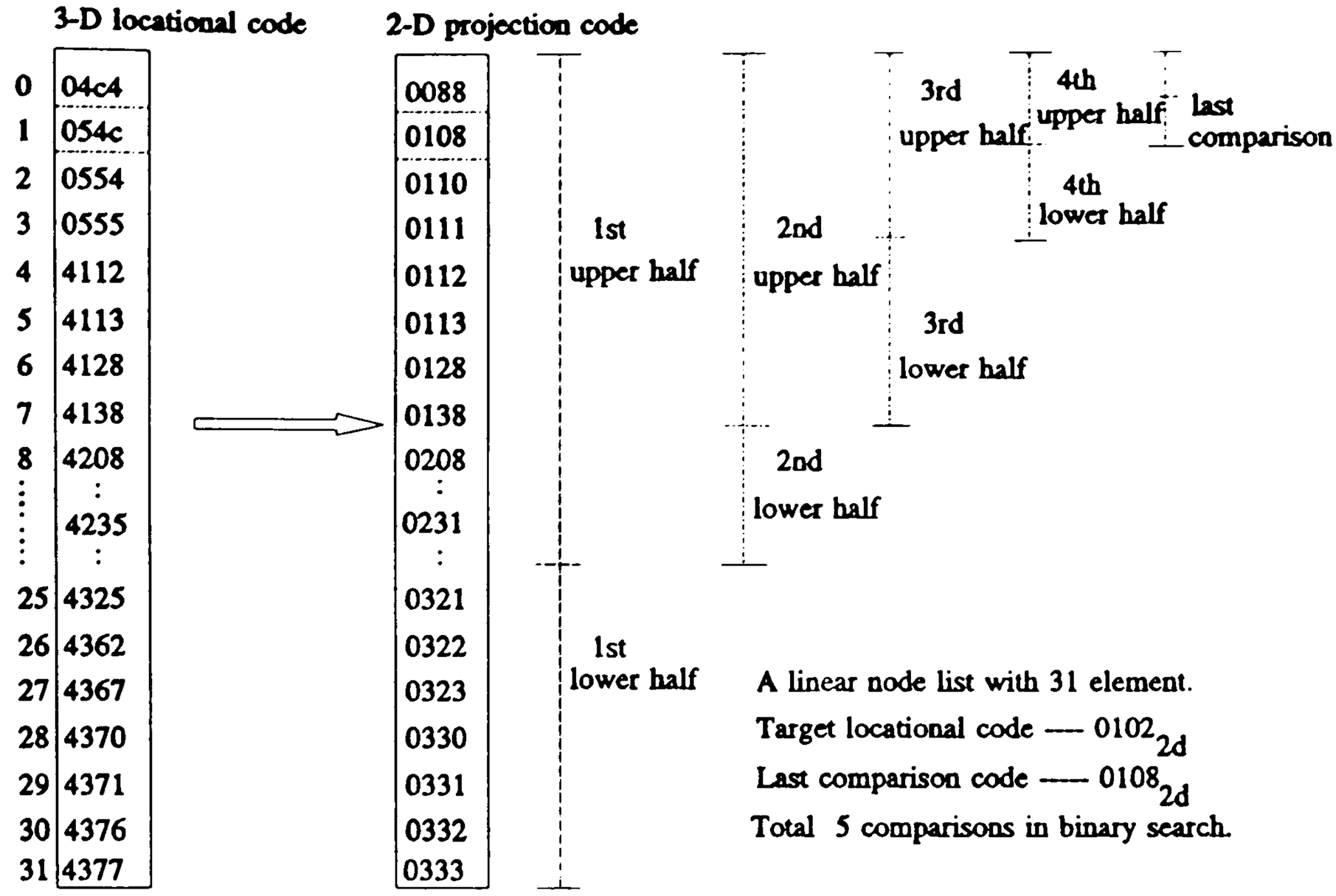


Figure 3.8 An example of accessing a node with 2-D locational code 0102.

```
procedure ACCESSING(value integer Node_list, N_L, *Nfound, *Nlocation)
begin
  value integer Top, Mid, Bottom, Key2D, Key3D;
  value integer N_L, *Nfound, *Nlocation, Flag;
  Top ← 1;
  Bottom ← N_L;
  *Nfound ← FALSE;
  *Nlocation ← 0;
  Key2D ← Key3D & 33333333_16; derive the projection code
  while target not found and not reach the end of current half element do
    begin redefine the half search space
      Mid ← RShift(Top+Bottom,2);
      Flag ← COMPARE(Key2D,Mid);
      if (Flag equals 0) the node found
      else if (Flag equals -1)
           node is in the upper half of the list;
      else if (Flag equals 1)
           node is in the lower half of the list;
    end;
  DECODING(*Nlocation);
  return (I,J,K,S);
end;
```

```
procedure COMPARE(Key2D, Mid)
begin
  value integer Mid, Key2D;
  value integer *Nlocation, Node_list[Mid], Mid2D;
  *Nlocation ← Node_list[Mid];
   Mid2D ← Node_list[Mid] & 33333333₁₆;
  if ((Key2D - Mid2D)) equals 0) node found;
    return (0);
  if ((Key2D - Mid2D)) less then 0)
    node is in upper half of the current half list;
    return (-1);
  else
    node is in lower half of the current half list;
    return (1);
end;
```

The accessing operation is based on a binary search of the terrain oct-tree. The time to execute the accessing is determined by the number of passes through the comparison loop. In each pass through the loop, the search involves at most half the elements that were under consideration during the preceding pass through the loop. Searching the array of sorted nodes can be performed in a time proportional to the logarithm of the total number of nodes [Aho74]. Thus in the worst case, the performance is $O(logN_L)$.

3.6.2 Adjacency

Adjacency is an important issue in oct-tree terrain structures. For example, as discussed in chapter four, the adjacency relations of a node can be incorporated in procedures to determine areas of 'danger'. A connected danger area is identified by recursively checking and extending the adjacent nodes of a given node until the boundary nodes of a connected danger area are reached. One important advantage of using terrain oct-trees is a possible reduction in the number of adjacent nodes.

In two dimensions, a node has a maximum of eight neighbours. Two nodes can be adjacent (and hence neighbours) along an edge or a vertex in four possible orientations. By contrast, in three dimensions, two nodes can be adjacent (and hence neighbours) along 6 possible faces, 12 possible edges or 8 possible vertices.

Although a terrain surface is a composition of 3-D spatial points, some adjacency relationships in 3-D space are meaningless in airborne terrain applications, for example where neighbours are beneath a peak point. Again, the one-to-one mapping relationship between a 3-D locational code and its projection code provides the ability to manipulate adjacent nodes in 2-D space.

Location of an adjacent node in a terrain oct-tree is performed in projection space. The co-ordinates of the adjacent points corresponding to a given point (I,J) in the main eight principal directions (denoted by N, S, W, E, SW, NW, SE, NE) are shown in Figure 3.9. Since both I and J are evaluated, determining whether or not a given pixel is on the border, corresponds to checking for $I,J=0$ and $I,J=$n-1. The following expansions give the arithmetic relationships:

$$
\begin{aligned}
N_{16} &= (I+(2*(J-1)_2)_{16})_{16} ; \\
S_{16} &= (I+(2*(J+1)_2)_{16})_{16} ; \qquad 3.14 \\
W_{16} &= (2*J+(I-1)_2)_{16} ; \\
E_{16} &= (2*J+(I+1)_2)_{16} ; \\
NE_{16} &= ((I+1)_2+(2*(J-1)_2)_{16})_{16} ; \\
SE_{16} &= ((I+1)_2+(2*(J+1)_2)_{16})_{16} ; \qquad 3.15 \\
SW_{16} &= ((I-1)_2+(2*(J+1)_2)_{16})_{16} ; \\
NW_{16} &= ((I-1)_2+(2*(J-1)_2)_{16})_{16} ;
\end{aligned}
$$

where the subscript denotes the number system used in the arithmetic operations.

For example, locating the SE adjacency node of locational code $Q_{2d} = 3000_{16}$ is achieved by firstly computing the values I and J of $Q_{2d} = 3000_{16}$ from equations 3-8 and 3-9 giving $J=1000_{16}$ and $I=1000_{16}$. Secondly, the adjacent projection code of Q in the SE direction is obtained by applying equation 3-15 to I, J giving SE = $(((1000+1)_2+2*(1000+1)_2)_{16})_{16}$. The arithmetic operations in this example are performed using a binary form, where the results are represented as base 16. Since the

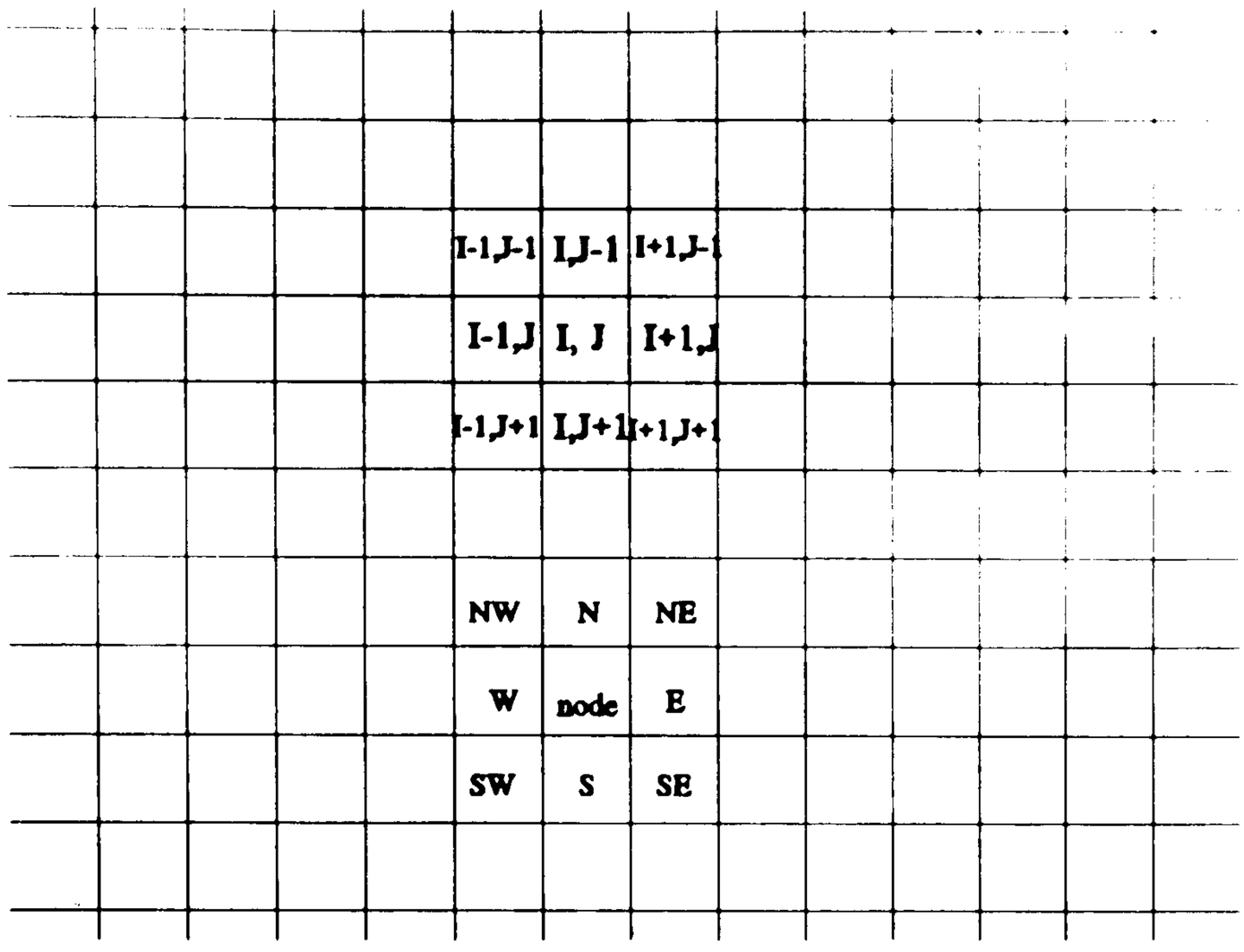


Figure 3.9 Adjacent points correspond to point (I,J) in principal direction.

locational codes are represented by hexadecimal numbers, many of the computations on the locational codes are implemented by means of bit manipulations such as shifting, masking, bit-wise logical operators and bit addressing through structures.

3.6.3 Neighbour Finding

The term *neighbour finding* is defined as a process that locates a node of size greater than or equal to a given node in specific orientations. When using a linear quad-tree representation, neighbour finding is a two-step process. Firstly, the address of an adjacent node is computed, then the list of nodes is searched for the specific node or for a larger node containing it.

A non-pixel level node is given as an example. The case of finding a node adjacent to Q, when Q itself is expressed as a merged node is treated by simply ignoring all '8's and 'C's, then appending those '8's and 'C's in the same bit positions in the translated node. For example, to find a node of equal size adjacent to $Q=3008_{16}$ in the westerly direction, $2*J=200_2$, $I=100_2$, and a temporary $Q' = ((200+011)_2)_{16} = 211_{16}$ is computed. Appending 8_{16} to 211_{16} gives the final value 2118_{16}.

In general, neighbours need not correspond to blocks of the same size. If the neighbour is a different size, it is initially processed as a node of equal size. After the projection code of an adjacent node is located, the binary search for the node using the projection code as its address will eventually locate the target node as described in the previous section.

In chapter four, the proposed oct-tree terrain model is applied to aircraft navigation simulation. As part of a Terrain Reference Navigation system, a flight path planning algorithm is devised.

CHAPTER 4

THE DESIGN OF A FLIGHT PATH PLANNING ALGORITHM

4.1 Introduction

Aircraft trajectories are formed from segments of climbing cruising, descending and turning flight which are selected by a pilot to achieve an ideal (or preferred) flight path. The problem of flight path planning involves developing preferred flight paths which avoid obstacles while satisfying general requirements of aircraft routing. An optimum solution of a collision free path in this context can be defined in terms of cost such as the shortest distance, fuel consumption or the exposure time of a path to a threat.

The problem of flight path planning is in many ways different from the case of path planning for robot:

- A mobile robot obtains information from the outside world using visual or auditory sensors and thus a mobile robot may have only an incomplete model of its environment. In flight path planning, the navigation environment is known to the navigation system, however, the obstacles need to be determined according to the given flight conditions (flight altitude, direction) before a preferable flight path can be found.

- When dealing with a large navigation space, path planning in robotics which is based on local sensory information may not lead to a globally good path. In flight path planning, the navigation capabilities are limited by the constraints of aircraft manoeuvrability and performance and thus global path planning is preferable.

- Although aircraft navigation implies three-dimensional motion, aircraft

navigation is often composed of straight line segments, in other words, there is no need to apply a 3-D path planning approach directly to the flight path planning problem.

- Most 3-D path planning approaches are based on small number of polyhedral obstacles in the navigation environment [Udup77, Loza79, Loza81] for relatively slow robot movement. Sharir has shown a very high computational cost (double exponential) of planning a collision-free path in three-dimensional space [Shar86], whereas the complexity of three dimensional path planning algorithms is at variance with high speed aircraft navigation which requires real time large database accessing.

The first and most important task in the development of an effective algorithm for flight path planning is the choice of an appropriate terrain representation on which the obstacles and free space can be clearly defined for path searching. In a TRN system, a DTED is partitioned into sub-blocks and individually compressed [Camb85, Chan85] to achieve the efficiency and speed requirements in a real time airborne environment. To determine a terrain profile, the height of the terrain at any point needs to be accessed individually from a DTED. For long distance flights, complete and accurate surface references are needed for the entire area of interest. The cost of accessing and retrieving DTED can be formidable.

As discussed in chapter two, the inability to exploit redundancy, the lack of flexibility and the inability to represent terrain at a coarser level of DTED lead to a hierarchical terrain representation. It also has been observed [Loza81] that the most important heuristic for a spatial representation of robot motion planning is to avoid excessive detail (and therefore time spent) on parts of the search space that do not affect the planning operation. A hierarchical structure has the ability to model the terrain with sufficient accuracy at a coarser level by truncating or approximating unnecessary data, allowing path searching task to be performed efficiently in real-time.

The representation of a terrain as a hierarchy of different levels of oct-trees enables data to be accessed at different resolutions, for example, where resolution is a function of distance from an aircraft. As mentioned in section 2.10, the use of a coarser level of a quad-tree can substantially reduce the number of nodes, thereby reducing the time complexity of a path searching process [Kamb86].

In this chapter, a flight path planning algorithm based on an oct-tree terrain model is presented. Although a terrain oct-tree represents a three-dimensional terrain surface, it can be projected onto two-dimensional space to simplify the path planning process. It demonstrates that the oct-tree terrain model is suitable for use in real-time airborne environment. In addition to producing effective flight planning methods, this work has focused on the inherent computational complexity.

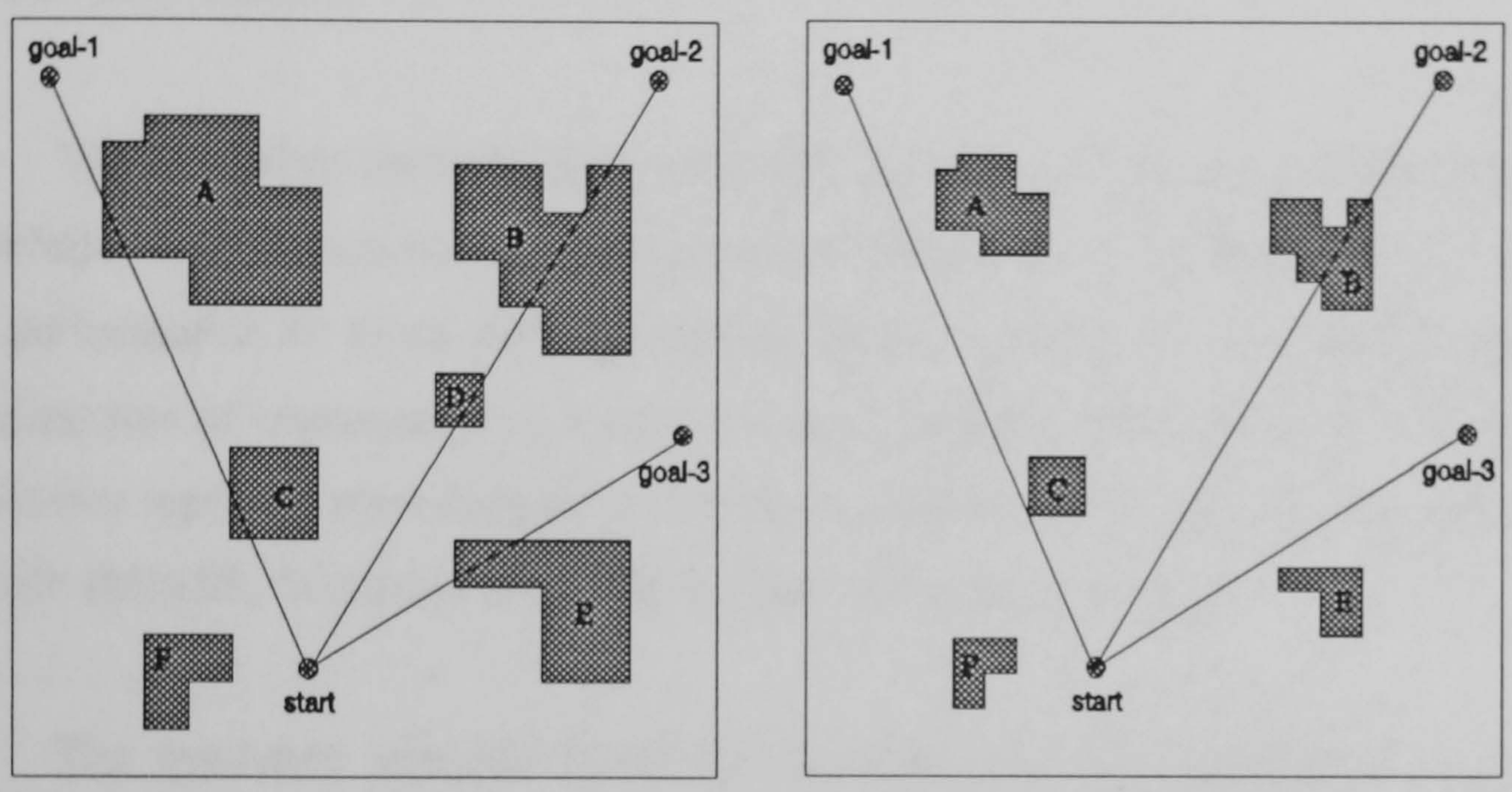
4.2 Flight Path Planning Approaches

Most path planning problems are based on the assumption that the mobile robot's location, desired destination and the location, size and shape of all obstacles in the terrain are known [Udup77, Loza79, Broo83]. In general, existing methods of path planning consist of two phases [Lato91]. In the first phase, a search space is generated. This search space consists of all the possible paths that will be considered and is represented by a graph. Once the search space is created, the second phase is to search for a desired path.

Lozano-Perez represents the obstacles in the navigation space as a visibility graph in which the shortest distance path can be found by searching the graph [Loza79]. O'Dunlaing and Yap represent a 'skeleton' of the free space known as the Voronoi Graph [O'dun82]. When traversing an edge, the path is collision-free since it maintains an equal distance between two adjacent obstacles.

However, prior approaches at dealing with predefined obstacles environment are not always compatible with the problem of flight path planning. The method described

above is appropriate to static and known environments where the navigation space and obstacles are predefined. For flight path planning, although the navigation environment is known to the aircraft (presuming that terrain elevation data is provided), there is no knowledge of the obstacles in the environment and obstacles must be acquired through sensory input.



(a) Obstacle regions A,B,C,D,E F with elevation above 1000 metres and the lines of sight between start and goal1, goal2, goal3.

(b) Obstacle regions A,B,C,D,F with elevation above 1500 metres and the lines of sight between start and goal1, goal2, goal3.

Figure 4.1 The visibility graph with respect to aircraft heading and safety altitude. The shaded areas represent the projection of obstacle areas.

An aircraft faces 'changing' sets of obstacles during navigation; the obstacles in a navigation environment may alter because of changes to a flight altitude, diversion to a new destination or the detection of a new threat on a predefined flight path. As illustrated in Figure 4.1, the visibility graph for path searching can also change as obstacles are varied. Each time a new goal point or a change in flight condition is given, a new set of obstacles in the navigation space must be abstracted from a terrain database for subsequent processing. In particular, during a flight mission, an aircraft may change to a new flight path several seconds after a request is initiated and therefore, the terrain data retrieval and flight path calculations must be performed and updated at relatively high rates. Different approaches are needed that can cope with an

environment of 'changing' objects.

Before a new visibility graph can be formed, it is necessary to determine the obstacles and map the obstacles to a configuration space. A new visibility graph is then constructed according to the new set of obstacles for the path searching process. The time cost to find a path should include the extraction of the obstacles in a real-time airborne environment.

Various other methods that subdivide the terrain into smaller units [Lato91, Kamb86] are also based on the assumption that the geometry of obstacles are invariant. The performance of those methods makes them unsuitable for a situation involving changing sets of obstacles. For instance, a path planning method which makes use of a quad-tree representation defines each node of a quad-tree as either an obstacle or non-obstacle [Mitc88, Wong86, Kamb86] (similar to a binary image).

The quad-tree schemes have the advantage that they aggregate a group of contiguous small squares into single large squares thus the path planning cost is reduced by using a smaller search space and searching at a coarser level of the quad-tree [Kamb86]. However, quad-tree schemes are not directly suitable for real-time flight path planning among changing obstacles due to the following observations:

- Firstly, in a quad-tree environment the obstacle and non-obstacle nodes are predefined. Once a quad-tree representation of navigation space is constructed, there is no simple way to update the obstacle areas with respect to a change of flight altitude unless the quad-tree is reconstructed. Although nodes can be inserted individually into a quad-tree, a change in the global representation caused by changing flight altitude is inevitable.

- Secondly, the quad-tree approach of finding a path is performed by expanding the path from a current position to the neighbouring free space. The search of free space among the current adjacent neighbours in a quad-tree results in a step-

wise path segment and is not suitable for high speed aircraft manoeuvring.

- Thirdly, the speed of the path searching is proportional to the number of nodes in a quad-tree [Kamb86]. For a quad-tree with a large number of nodes, the search for a path among free nodes is not efficient enough to match the real-time navigation requirements.

In the following sections, a path planning method is proposed which combines a digital model representation (grid, quad-tree) of terrain oct-trees with the searching space of a visibility graph used in the geometry model (vertices, line, polygon) [Loza79]. The algorithm is a configuration space modelling scheme for obstacles based on terrain oct-tree representation. The obstacles are transformed and represented explicitly as vertices of polygons, and the free space is defined implicitly by being outside of those obstacles. A visibility graph thus represents 'all' the visible paths between vertices of obstacles and start and goal locations for path searching.

4.3 The Modelling of a Navigation Space

The major purpose of a real-time flight path planning algorithm is to provide an aircraft with a new path in response to a new navigation situation. The aircraft must avoid all obstacles which are located in the current navigation space according to a constraint of flight altitude or predefined danger areas. The algorithm must be capable of updating a path within a few seconds of a request and this updating of the flight path may occur frequently during navigation. The principal objective of flight path planning is to generate a flight path by first "exploring" the obstacles and then specifying a set of waypoints which the aircraft will overfly in succession. The term "exploring" is used to emphasize a real-time navigation environment in which the obstacles need to be determined.

In order to simplify the problem at this stage, a flight path planning algorithm is described based on the concepts of true altitude and absolute altitude [Siou93]. The definitions of altitude are as follows (see Figure 4.2):

- **Absolute altitude** (H_{abs}) The absolute altitude of an aircraft is defined as the height above the surface of the earth at any given surface location. The vertical clearance between an aircraft and a mountain top is an example of absolute altitude.

- **True altitude** (H_{true}) The true altitude of an aircraft is defined as the actual height above standard sea level. True altitude is the sum of absolute altitude and the elevation above sea level of the ground below the aircraft. The true altitude is also termed **flight altitude** in this thesis.

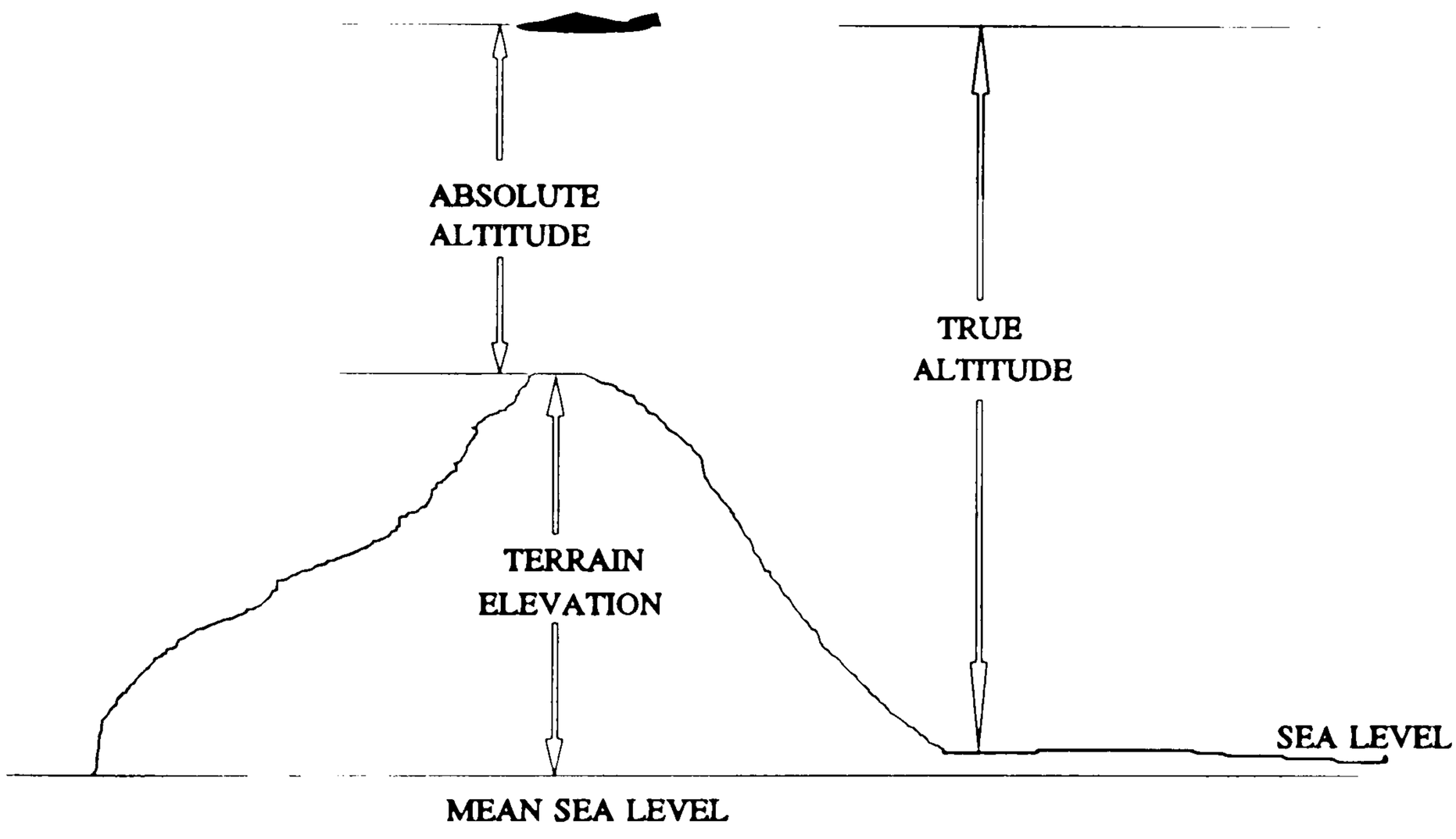


Figure 4.2 Absolute and True altitude of aircraft.

If both the true altitude and the absolute altitude are fixed, the difference between H_{true} and H_{abs} *is* termed *safety altitude*. The true altitude is used as a criteria to determine vertical distance from the obstacles from a terrain oct-tree, consequently, the navigation space is divided into *free space* and *obstacle space*.

The flight path planning algorithm comprises three stages:

Stage 1. the acquisition of obstacles.

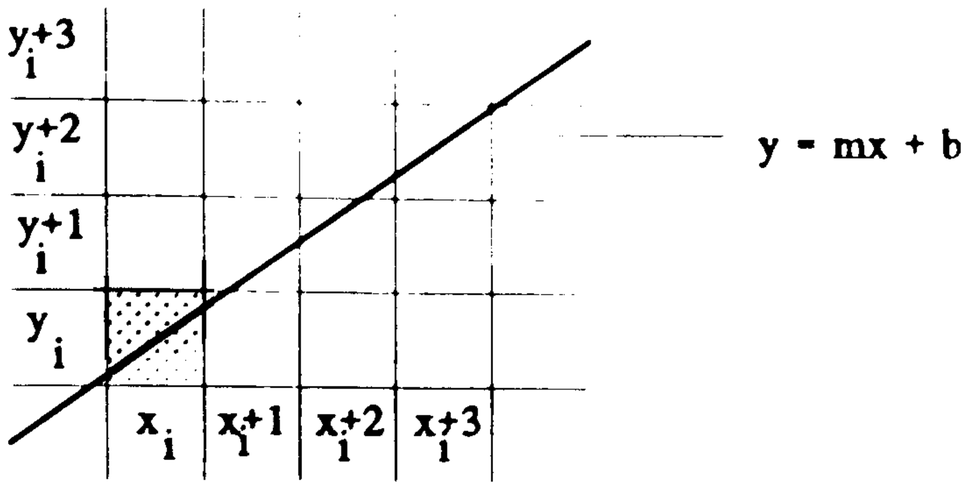
Stage 2. the transformation of obstacles to a search space.

Stage 3. the extraction of a flight path.

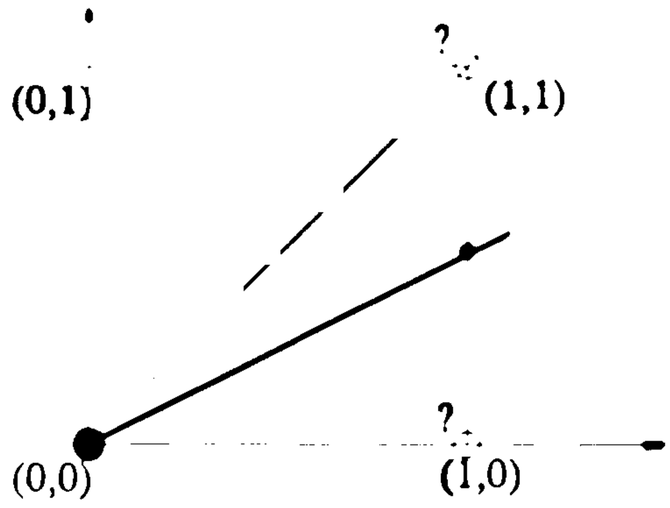
In the acquisition stage, the obstacles in the direction of the flight path are extracted from a given terrain oct-tree model. In the transformation stage, an aircraft is considered as a single moving point and the obstacles are assumed to be a collection of polyhedral bodies. The algorithm transforms the obstacles so that they represent the locus of forbidden positions for the moving point. A path of this moving point which avoids all forbidden regions is free of collisions. Paths are found by searching a visibility graph which is formed from the vertices of the transformed obstacles.

Rather than defining objects by geometrical data (such as elevation, area, co-ordinates of obstacles), terrain oct-tree models (described in chapter three) are used to represent the navigation space of an aircraft by using an oct-tree; no other data structure of features (linear or polygonal features) is required to describe the navigation space. The terrain oct-tree is provided in the form of a linear list in which each element represents a leaf node of the tree. A node is represented by its 3-D locational code in integer format. A 3-D locational code of a node contains the planar co-ordinates of north-west corner, a scaled elevation and size information of a large homogeneous block region, as described in section 3.3 (Figure 3.5).

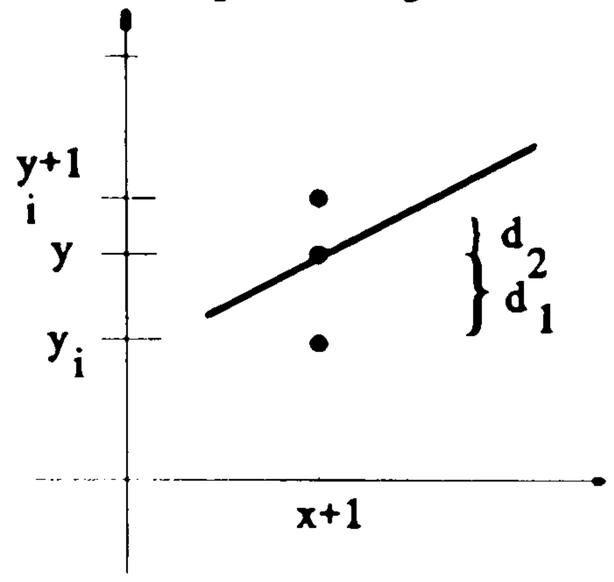
In addition to the terrain oct-trees which are used to represent region data, points and lines can also be represented by locational codes to provide locations of waypoints and flight paths in path planning. The representation of point and line data in locational codes are identical to that of a region representation. For example, a point data such as start or goal position of a flight mission is represented by both its locational code and is treated as a single unit leaf node. Linear features are represented by means of a set of consecutive points. For example, a path segment is obtained by applying the Bresenham's algorithm to its start and end points to determine the component points



(a) a sample line segment.



(b) two possible next points.



$d_1 = y - y_i = m(x_i + 1) + b - y_i$

$d_2 = (y_i + 1) - y = y_i + 1 - m(x_i + 1) - b.$

$d_1 - d_2 = 2m(x_i + 1) - 2y_i + 2b - 1.$

(c) determining which point closer to (x+1,y).

Figure 4.3 The Bresenham's line generation algorithm.

between them, as shown in Figure 4.3. The consistency of feature representations allow the path planning operations such as collision checking and line segment generation to be performed in a straightforward way.

A oct-tree terrain model has both two dimensional and three dimensional features in which a node in two-dimension space is a one-to-one projection from a node in three-dimension space. These features allow the three-dimensional flight path planning process to be computed in two-dimensional space. As discussed in chapter three, the 2-D and 3-D locational codes in a terrain oct-tree have a one-to-one mapping relationship; the difference between them is that the K value (the $(4^{n-1}+2)$th bits) is ignored in 2-D locational codes. The use of two-dimensional representations simplify such tasks as locating an adjacent point or defining a path segment between two waypoints during the planning process. The 2-D locational codes result in a binary representation of navigation space where a node is either a free space or an obstacle node. The path planning process thus proceeds in two-dimensional space.

4.4 The Extraction of Obstacles

4.4.1 Exploring the Obstacles

Terrain oct-tree representation and the basic requirement of flight planning were outlined in the previous section. Initially, a terrain oct-tree is simply a list of integers where each integer represents a square area of common scaled elevation. The list is stored in an ascending sequence which results from the Morton sequence encoding of a terrain matrix. The first stage of the algorithm is to extract the obstacle nodes from a terrain oct-tree and to reconstruct them in the configuration space of an aircraft.

Nodes in a terrain oct-tree with scaled elevation values above a safety altitude are defined as "*danger nodes*". A danger nodes list is similar to the linear quad-tree representation of a binary image in which only the black nodes are stored [Garg82a]. In the proposed algorithm, most operations are performed on the danger nodes (black nodes) and the speed of the subsequent planning process thus depends on the size of the list of danger nodes. As a terrain oct-tree is retrieved from secondary storage, the danger nodes list is generated according to a pre-defined safety altitude and is stored as a list in an ascending sequence.

The list of danger nodes does not give any topographic information of the obstacles area directly. The nodes contain no information about their geographical relationship (e.g., connectivity, boundary) in the navigation space. For example, a danger area containing 26 nodes and organised as an ordered list is shown in Figure 4.4. It is not apparent from the list whether node 1 or node 26 initially belong to the same connected region. Moreover, the physical locations of the danger nodes are 'scattered around' the navigation space and clearly, only a portion of the danger nodes are likely to jeopardize the track of the aircraft. For instance, in Figure 4.5, there are five connected danger regions in the navigation space, but if an aircraft is flying to goal 1, the possible obstacles are limited to regions A and C.

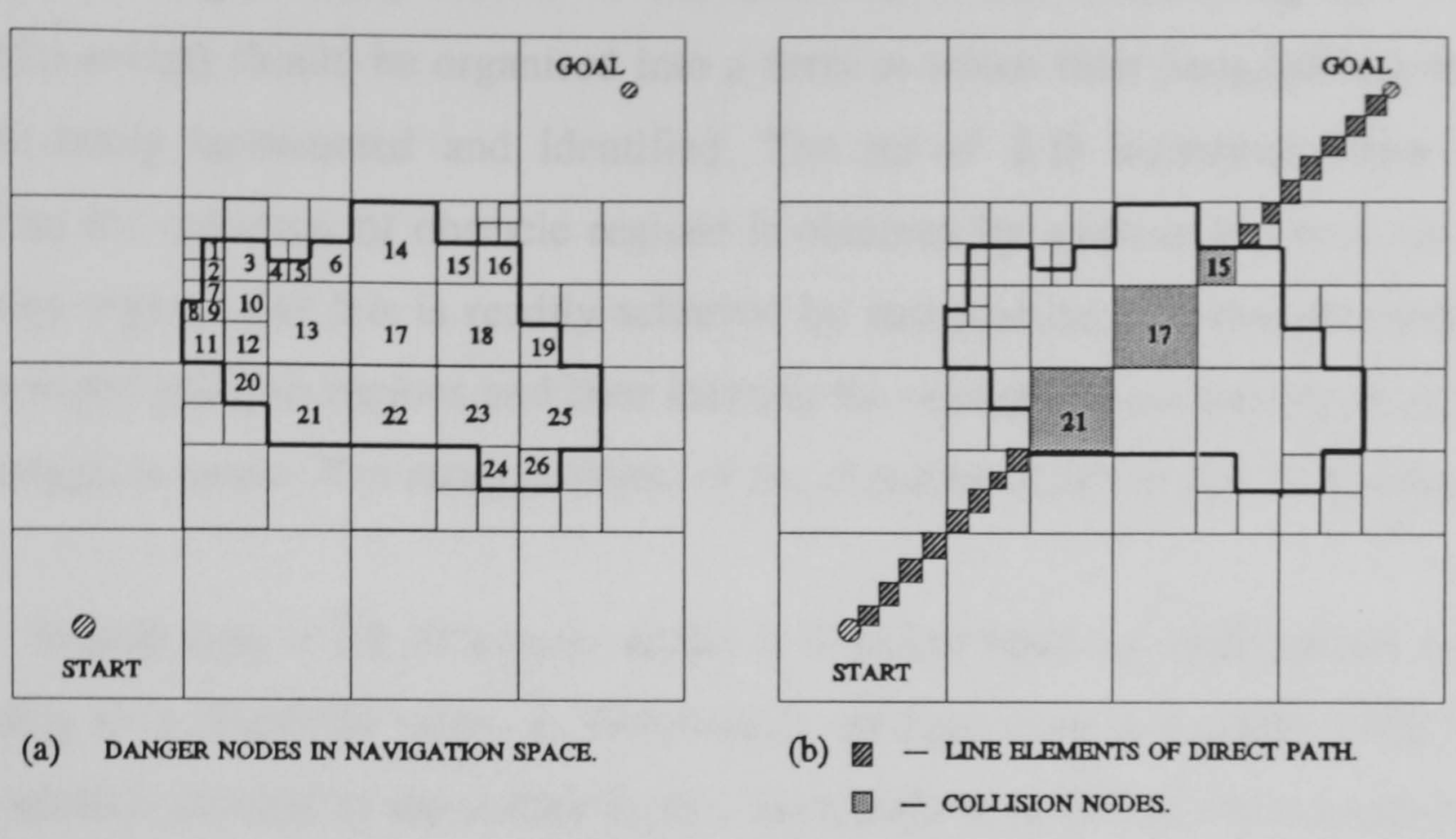


Figure 4.4 An obstacle area consists of 26 danger nodes.

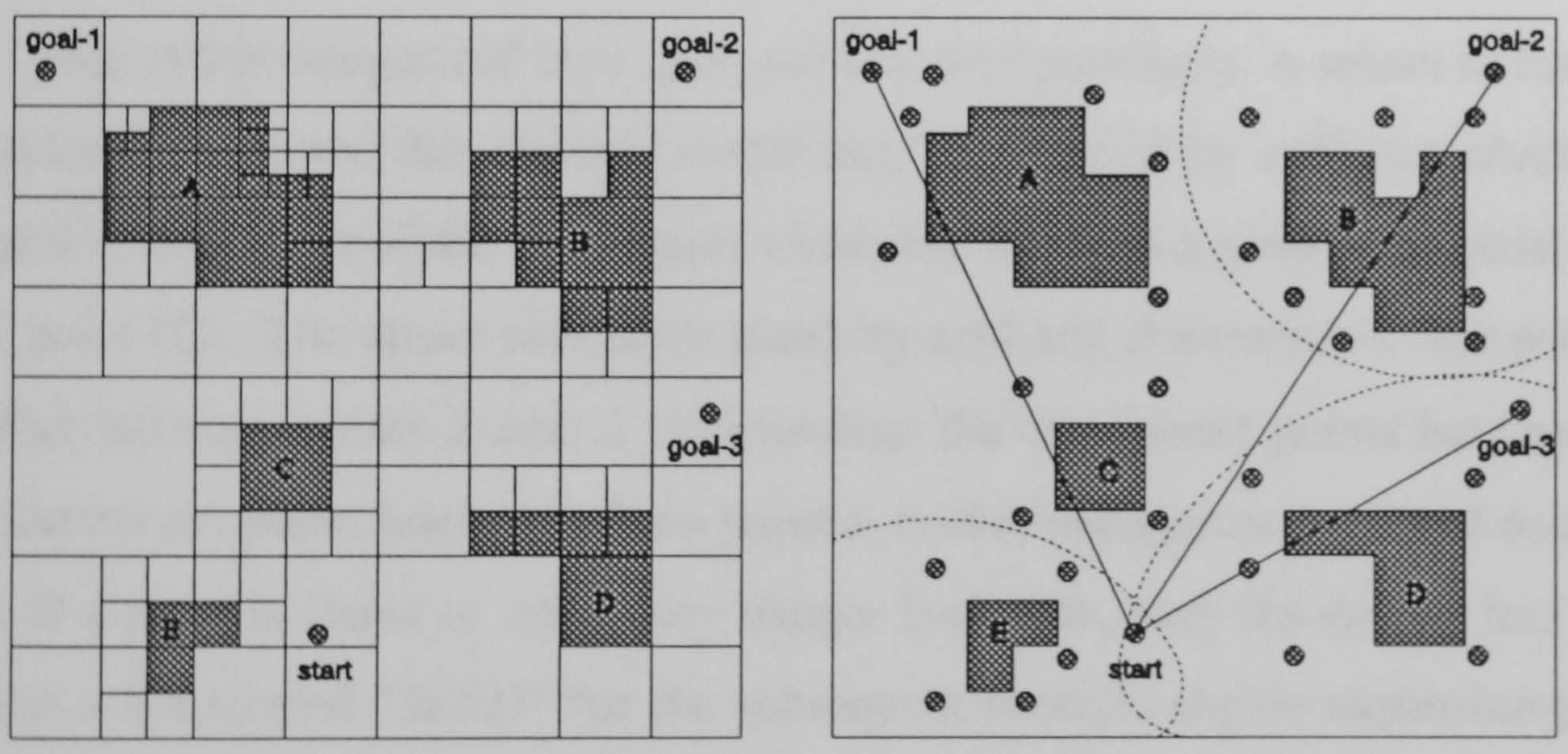


Figure 4.5 The possible obstacle regions with respect to direction.

The danger nodes related to the direction of the current segment (termed *obstacles nodes*) should be organised into a form in which their geographical features can be easily represented and identified. The set of 2-D locational codes which represent the coverage of obstacle regions is obtained by locating the vertices of the obstacles regions and this is readily achieved by reorganizing the obstacles into a set of connected polygon regions and then locating the vertices of each polygon region in the navigation space. The reorganisation of the obstacles involves the following steps:

In step one, a list of *danger nodes* is obtained from an input terrain oct-tree according to a *threshold* value. A *threshold* is obtained from a rounded value of the safety altitude divided by the scaling factor. Each node in an oct-tree is examined to see if its K value (embedded in the $(4^{n-1}+2)$th bits of its locational code) exceeds the threshold. Nodes with K values less than the threshold can be ignored. The resultant list contains all the danger leaf nodes in the aircraft navigation space. Figure 4.4a is an example of a navigation space with danger leaf nodes illustrated. The number attached to each danger leaf node indicates the sequence of the danger nodes in the list.

Step two is categorised by a 'generate and test' paradigm. A subset of the danger leaf nodes list, termed the *obstacle nodes list*, is obtained by *collision checking* (or *intersection detection*) of the direct path which lies between a given start point (*S*) and a goal point (*G*). The direct path is obtained by applying Bresenham's line generation algorithm between points *S* and *G* to determine the component points between them. Each component point, beginning from point *S*, is checked against the list of danger leaf nodes. If a point is found to match any danger leaf node, then the danger leaf node is stored in a list termed "*SEED*" for the subsequent obstacle region expansion process. Figure 4.4b shows the intersection between a direct path and a set of obstacles nodes.

The motivation for performing the collision check of a direct path between the points *S* and *G* is based on the fact that the shortest path between two points is a direct line. If a direct path is obstructed by obstacles, then a detour is taken to avoid the collision. This checking procedure is not completed until all the points along the path

have been checked. If a point is found to collide with a danger leaf node, then it is either exactly matched with that node or it is enclosed by that node. Sometimes, more than one component of a point along a direct path will collide with the same obstacle node. For instance, in Figure 4.4 obstacle node 21 encloses 4 point elements, whereas node 15 enclosed 2 point elements. An obstacle node must be checked to establish if it is already in the *SEED* list before it can be appended to it.

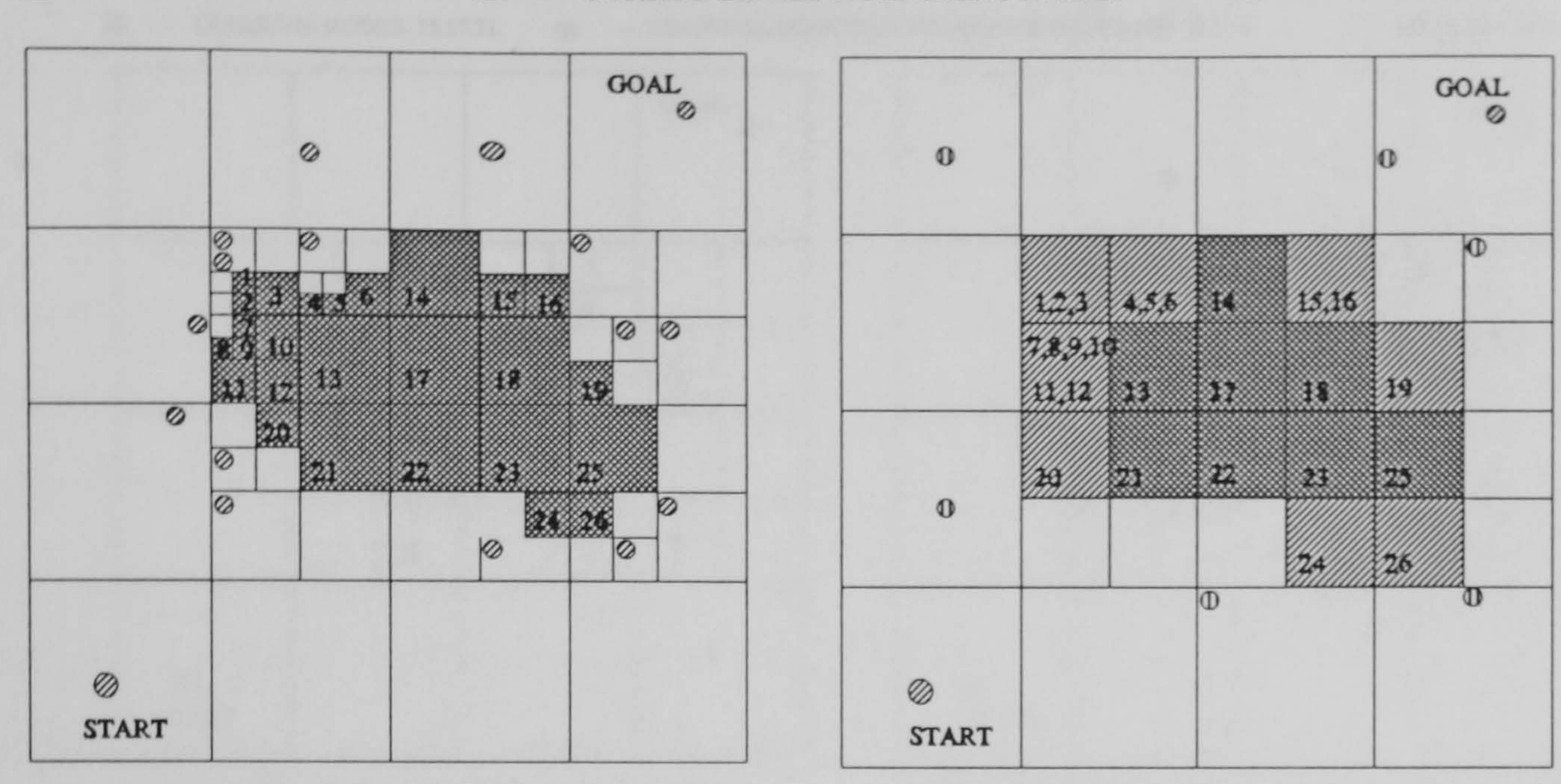
In step three, each obstacle leaf node in the *SEED* list (generated in step two) is used as a 'seed' to 'grow' a connected obstacle region and hence to locate the vertices of the region. An obstacle region expansion process involves finding the obstacle leaf nodes adjacent to the leaf node being expanded. The major purpose of this expansion process is to obtain a set of *waypoints*[•] which correspond to the obstacle region in the navigation space where the waypoints are used as possible detour points of a flight path to avoid the collision.

The obstacle leaf node is expanded towards its neighbours in four main directions (neighbours in the horizontal and vertical directions) and this operation is performed recursively until a *boundary node* is reached which does not have neighbouring nodes in the list of danger leaf nodes. Figure 4.6a shows the expansion of an obstacle region. During the expansion process, the *boundary type* of a boundary node is obtained and is used to determine whether it is a vertex of an obstacle region. If it is a vertex node, a *waypoint* is then located in the diagonal direction outside the obstacle region (also shown in Figure 4.6b).

The size of a neighbouring node may be different from the size of an 'expanding' node (the size of a 2^dx 2^d node is defined to be 2^d and the level of the node is defined to be d). This expansion process is performed by computing the locational codes of neighbouring nodes in the four main directions, one direction at a time (north,

---

• The term waypoint is not used in the strict navigation sense of being predefined turning point in a flight path.



(c) THE EXACTLY LEVEL OF BOUNDARY NODES WITH 17 WAYPOINTS.

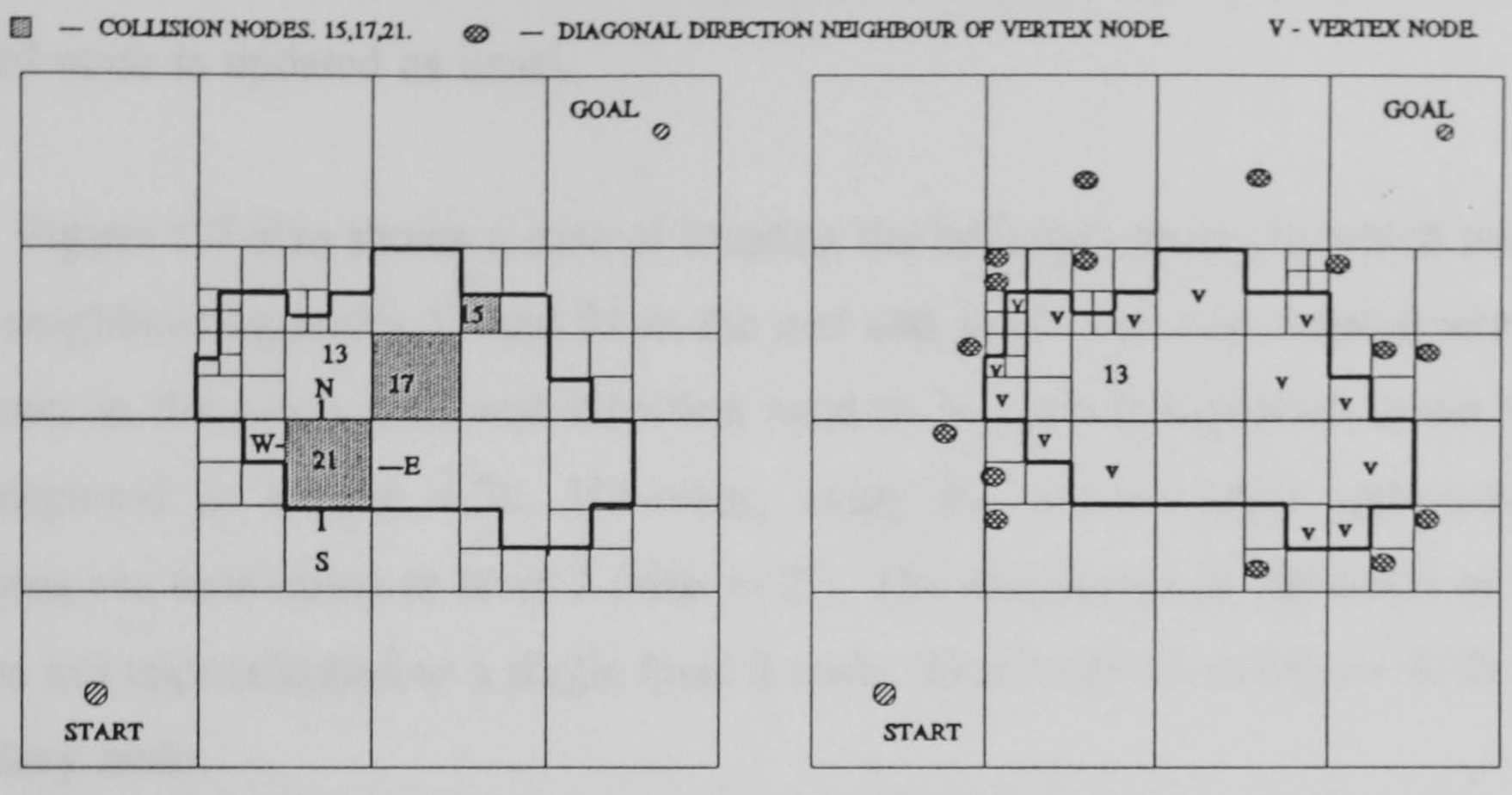
(b) THE COARSER LEVEL OF BOUNDARY NODES WITH 8 WAYPOINTS.

Figure 4.6 An example of obstacle region expansion and waypoint derivation.

east, south, west) to determine if it is a boundary node. The locational code of a neighbour node is searched against the danger nodes list for an exact match starting from a neighbour of equal size. If an equal size neighbour is not found in the list, the neighbour with a larger size is searched and this process is repeated until a neighbour is found or the level next to the root is reached.

If a neighbouring node is still not found after this process, it implies that either some smaller size neighbouring nodes may exist or the node is a boundary node. As described in section 3.6, a binary search is terminated as soon as it is apparent that the node is not present. However, if the node is not present in the nodes list, it may be contained in a merged node of an upper level. The node with the projection code which is used in the last comparison loop actually covers this queried node. This feature is also used to determine if there are any quadrants of the same size as the neighbouring node. If the locational code which is used in the last comparison loop is covered by the queried neighbouring node, a further search is necessary, otherwise the currently expanded node is a boundary. For example, node 21 in Figure 4.7a needs further

processing in the west direction and encounters a boundary in the south direction.



(a) EXPANSION FROM NODE 21 TO ITS FOUR MAIN DIRECTION NEIGHBOURS RECURSIVELY.

(b) WAYPOINTS ARE LOCATED ACCORDING TO THE VERTICES.

Figure 4.7 An example of obstacle nodes expansion process.

Two further approaches that either reduce the size of the expansion node or use coarser level node approximations can be applied :

The first approach is implemented in the case that detailed information is required. It is achieved by subdividing the currently expanded node and expanding each subquadrants individually. In this case, the currently expanded node is treated as a boundary node and its boundary type is updated. This process is performed recursively until either a neighbouring node is found or the process reaches a boundary node (ie., the neighbouring node is not found) at the pixel resolution level.

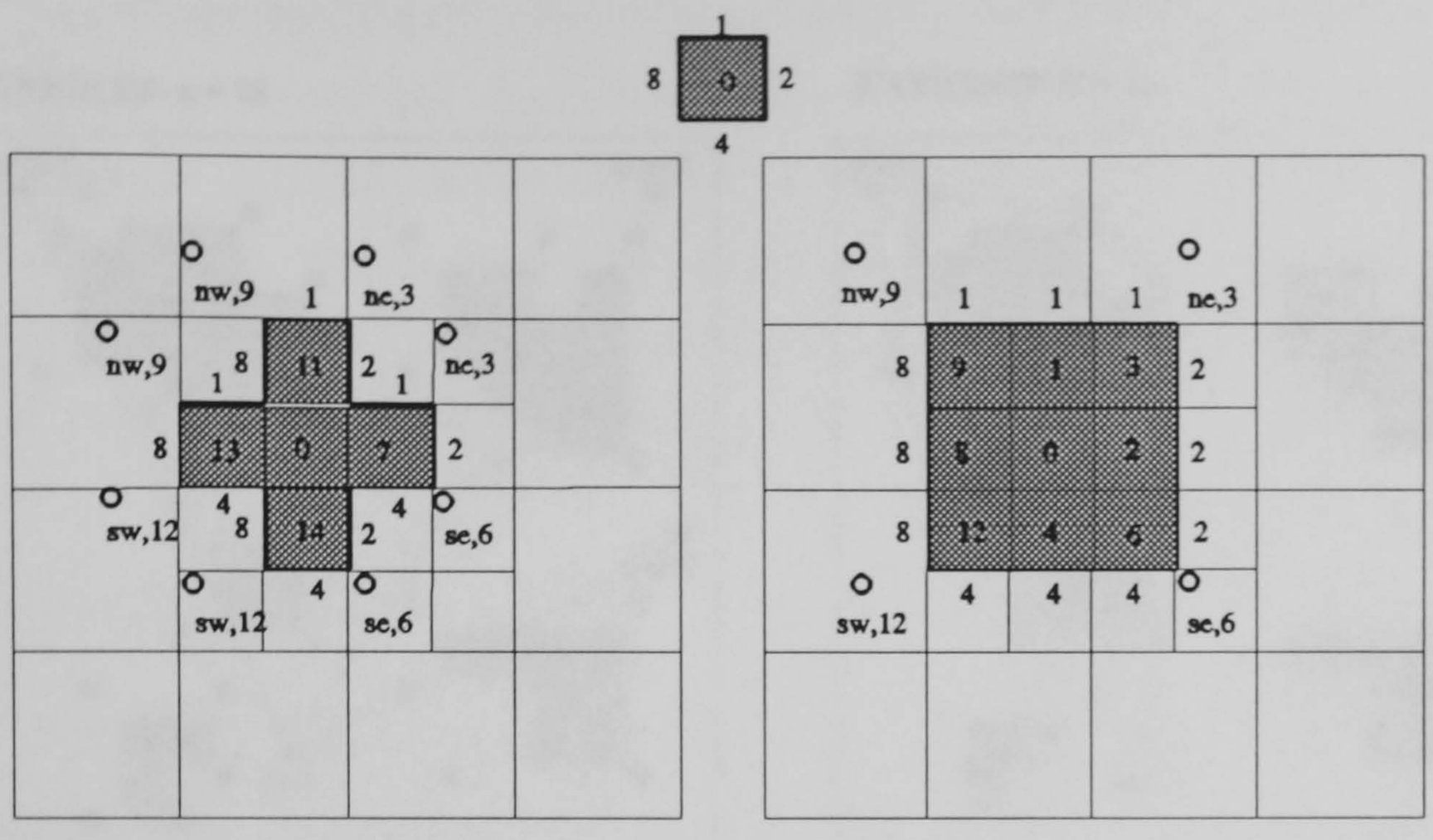
In the second method, the hierarchical features of a terrain oct-tree are exploited. In order to make this process efficient and to reduce the number of vertices (and hence reduce the size of the visibility graph), instead of reducing the size of the currently expanded node, the locational code of an equal sized neighbouring node is assigned as a boundary node and the expansion process is terminated. This is an

approximation process that truncates those nodes below the currently expanded resolution level to reduce the number of vertices. The boundary code of the currently expanded node is updated as usual.

Figure 4.7 also shows a case of locating the boundary nodes in which node 13 locates neighbouring nodes 17 and 21 in the east and south direction respectively. The neighbours in the north and west direction need to be further expanded down to the levels depicted in Figure 4.7a. However, using the coarser level approach, the expansions are terminated at level 2 (size = 2^2). The neighbours in the north or south direction are approximated to a single level 2 node. Thus node 13 in Figure 4.7b is not a boundary node.

Whenever a neighbour is found, it means that further expansion is needed in that current direction, otherwise the node has encountered a boundary node. After all the four main directions are examined, the boundary type of an obstacle node is obtained. From the boundary type, it can be established if the obstacle node is a vertex node. The expansion process proceeds recursively from the neighbouring nodes. An obstacle nodes list is maintained containing the nodes which have been expanded because it is necessary to check the list of obstacle nodes to prevent the expansion process from examining the same obstacle nodes over and over again.

An encoding method is used to indicate the boundary type of an obstacle node if any part of an obstacle node is a boundary of an obstacle region. Zero indicates that a node does not have any side that is on the boundary. The codes 1, 2, 4 and 8 correspond to the sides that are on the northern, eastern, southern and western boundaries, respectively. These codes are additive. Figure 4.8 shows the possible combinations of boundary type. A node with a boundary type other than 0, 1, 2, 4, 8 is defined as a vertex node of an obstacle region. For example, a node whose sides form the southern and western boundaries has a code of 12, whereas a node with boundary code 7 has north, east and south sides in the boundary.



O DIAGONAL NEIGHBOURS OF VERTICES NODES, POSITION OF WAYPOINT.

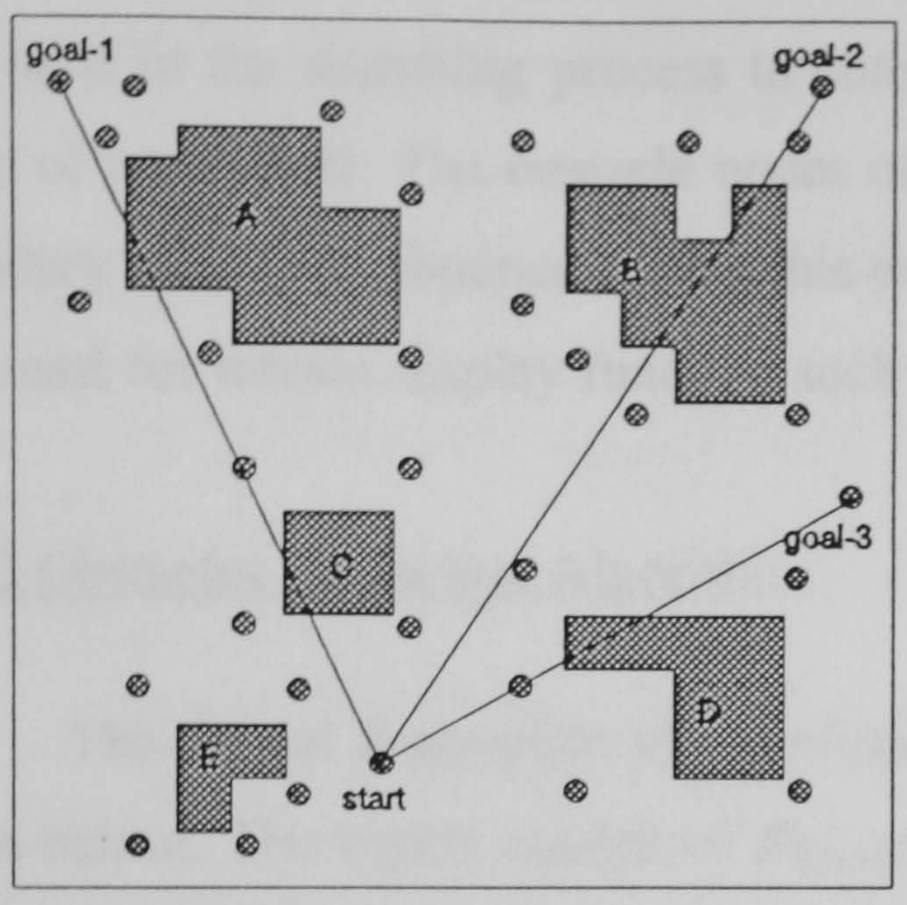
NW,NE,SW,SE REPRESENT DIAGONAL DIRECTION NEIGHBOURS.

Figure 4.8 The combination of boundary type and waypoint position.

By means of this boundary type representation, the waypoints are obtained which extend to the obstacle region. Figure 4.8 also shows the possible positions of waypoints. For example, a node with a boundary code of 13 is a vertex node. It has two waypoints in the NW and SW diagonal direction. A waypoint is represented by the locational code of the north-west corner of the neighbours in the diagonal direction and is stored in a list. Once again, it is necessary to check if a waypoint is not a member of the list of danger nodes and does not already exist in the waypoints list, before it can be appended to the list of waypoints.

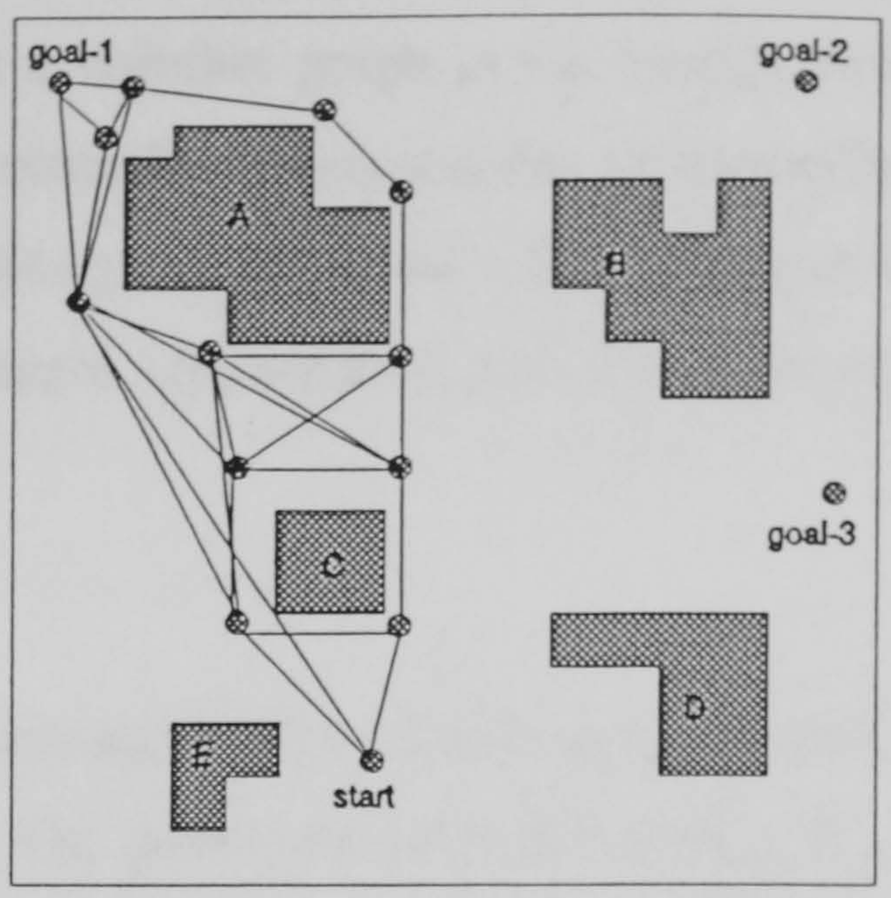
After the expansion process has been applied to all the members of the *SEED* list, the obstacle regions are obtained along a direct path between the start and goal points dependent on the aircraft's direction. Figure $4.9_{a\text{-}d}$ shows a gaming area with danger nodes and corresponding regions of obstacles along different direct paths between the start and goal nodes.

VERTICES n = 28



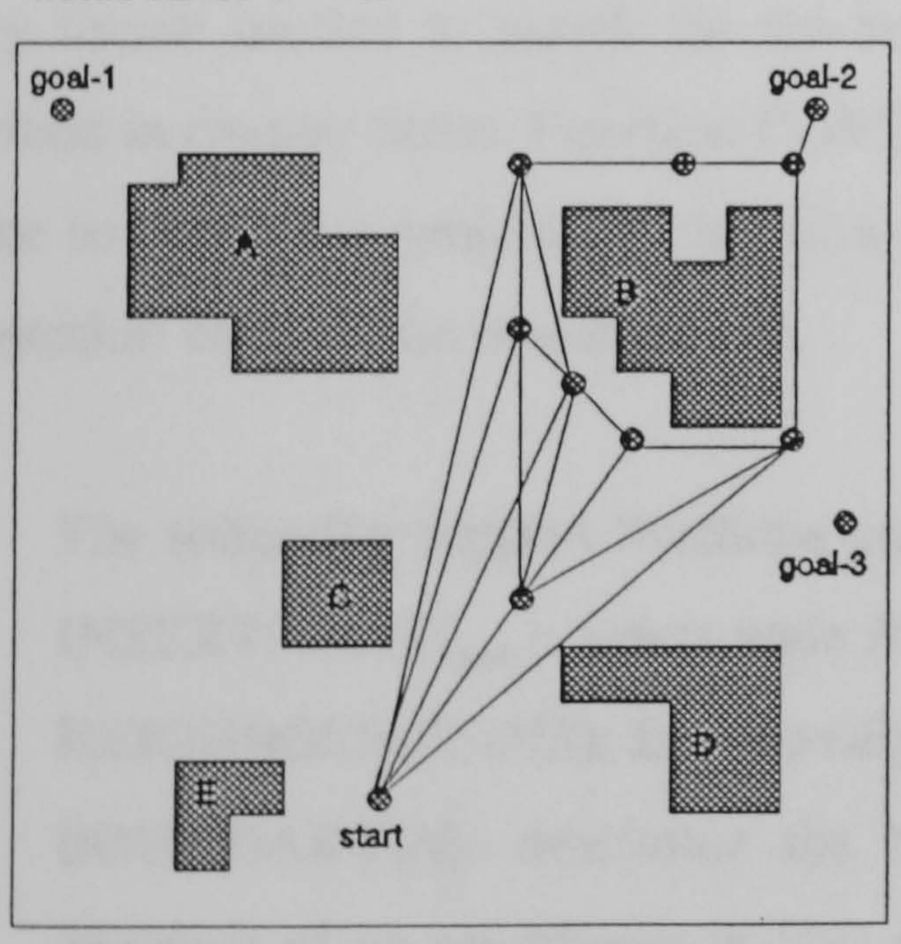
(a) A NAVIGATION SPACE WITH 32 WAYPOINTS.

WAYPOINTS W = 13



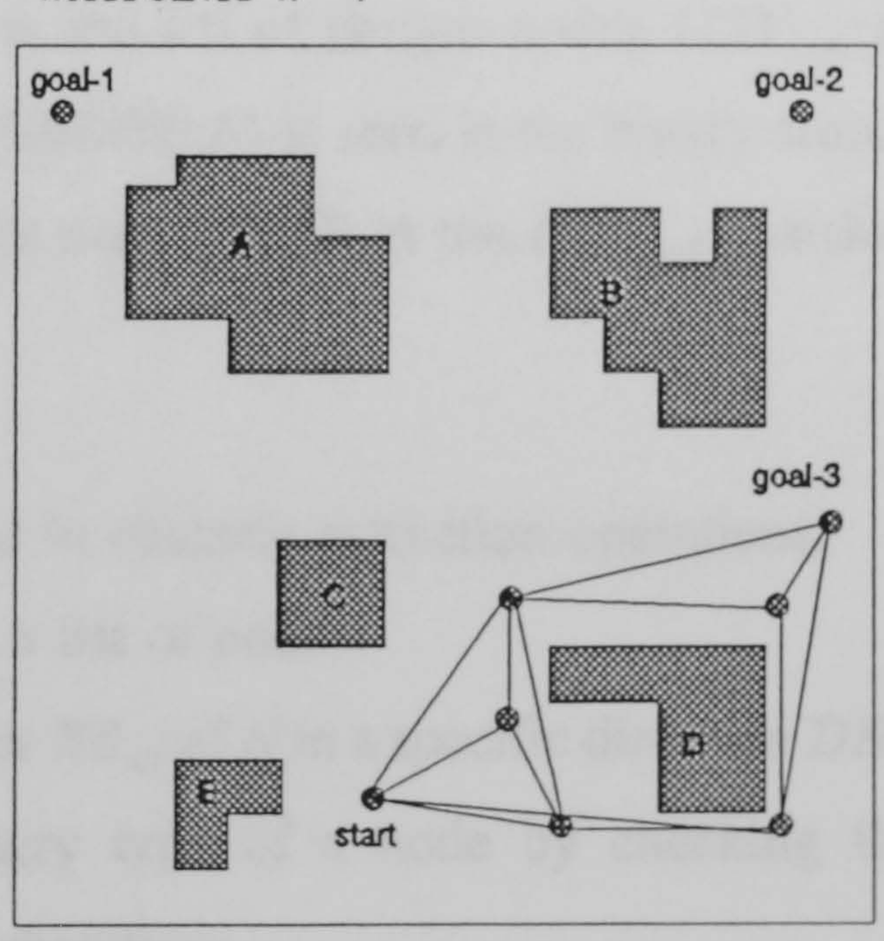
(b) PATH START TO GOAL-1, 13 WAYPOINTS.

WAYPOINTS W = 10



(c) PATH START TO GOAL-2, 10 WAYPOINTS.

WAYPOINTS W = 7



(d) PATH START TO GOAL-3, 7 WAYPOINTS.

Figure 4.9 The obstacle regions in a navigation space with respect to different start and goal points.

The actual topographic information of the obstacles regions is abstracted from the given terrain oct-tree and transformed into a set of waypoints. These waypoints are then used in the searching process to construct a visibility graph in the configuration space of an aircraft. The obstacle nodes of the obstacle regions and the corresponding boundary nodes are obtained during this expansion process and the information can be also used for terrain display function such as danger area masking and obstacle cuing.

4.4.2 Obstacles Extraction Algorithm

The formal description of the obstacles extraction algorithm in pseudo codes is given below. The inputs consist of N_{start}, N_{goal}. The procedure GET_SEED(N_{start},N_{goal}) uses Bresenham's line generation algorithm to get a list *SEED* of collision nodes. The procedure OBSTACLE_GROWING(*SEED*) extracts the connective regions of obstacles from the list of danger nodes. Accordingly, the waypoints in the diagonal directions of the vertices of the obstacle regions are located. Procedure SEARCH(N,$LIST_{node}$) is a binary search routine to search for the key N in the list of danger nodes $LIST_{node}$ as described in chapter three. Function COMPARE(*NODE*,*N*) is used in the binary search routine to check if a projection code of a danger node *NODE* in the $LIST_{node}$ matches a projection code of the search key N.

The following support functions are used in obstacle extraction operations:

- INSERT(N,$LIST_{node}$): insert node N into a list of nodes.
- NEIGHBOUR(N,*DIR*): find the neighbour NB_{dir} of N in a specific direction *DIR*.
- BOUNDARY(N): determine the boundary type of a node by checking the presence of its neighbours in four main directions.
- VERTEX(*NODE*): according to the boundary type of a node, determine the locational code of its diagonal vertices.
- Rshift/Lshift(V,S): bit operations to shift V right/left by S bits.

```
procedure GET_SEED(value integer N_start,N_goal);
begin
  value integer Top, Mid, Bottom;
  value integer *Nfound, *Nlocation, Flag;
  Top ← 1;
  Bottom ← number of nodes in danger nodes list;
  *Nfound ← 0;
  *Nlocation ← 0;
  for Every point element E_point obtained from BRESENHAM(N_start to N_goal) do
begin
  while Top < = Bottom and *Nfound equals 0  do
   begin
      Mid ← RShift (Top+Bottom,2);
      Flag ← COMPARE(E_point,Mid);
      if Flag equals 0, exactly match then
         begin
            *Nfound ← 1;
            E_point ← 2D locational code of *Nlocation;
            OBSTACLE_GROWING(E_point);
            if E_point not in the SEED list then
              INSERT(E_point, SEED);
         end;
      else if Flag = -1 then
         Bottom ← Mid - 1; /*for next loop comparison*/
       else if Flag = 1 then
         Top ← Mid + 1; /*for next loop comparison*/
    end;
  if *Nfound = 0 then
    begin
      if E_point enclosed by *Nlocation
         begin
            OBSTACLE_GROWING(E_point);
            E_point ← 2D locational code of *Nlocation;
            if E_point not in the SEED list then
               INSERT(E_point, SEED);
         end;
    end;
end;
end;

procedure OBSTACLE_GROWING(value integer E_point)
begin
  value integer Top, Mid, Bottom, Direction;
  value integer *Nfound, *Nlocation, Flag;
  Top ← 1;
  Bottom ← number of nodes in danger nodes list;
```

```
*Nfound ← 0;
*Nlocation ← 0;
get size information S of E_point
for each NEIGHBOUR(E_point,Direction) = NB_dir not already in the obstacles list do
   begin
      while Top <= Bottom and *Nfound = 0 do
        SEARCH(NB_dir);
      if *Nfound = 0 and the pixel level is not reached
        begin
      if NB_dir does enclose any danger node /*determined by binary search*/
      case 1 refinement of exact level of boundary nodes
        begin
          subdivide E_point into four quadrant
          for each quadrants do
             OBSTACLE_GROWING(QUADRANT)
        end;
      case 2 coarser level approximation
        begin
          BOUNDARY(E_point); update boundary type of current node;
          NB_dir ← adjacent node of *Nlocation;
          INSERT(NB_dir,OBSTACLE);
          BOUNDARY(NB_dir);
          VERTEX(NB_dir);
        end;
            else encounter a boundary, update boundary type
                BOUNDARY(E_point);
          end;
        if *Nfound = TRUE;
          OBSTACLE_GROWING(NB_dir);
      end;
    VERTEX(E_point);
end;

procedure SEARCH(value integer E_point)
begin
  value integer Top, Mid, Bottom;
  value integer *Nfound, *Nlocation, Flag;
  Top ← 1;
  Bottom ← number of nodes in nodes list;
  *Nfound ← 0;
  *Nlocation ← 0;
  Mid ← RShift (Top+Bottom,2);
  Flag ← COMPARE(E_point,Mid);
  if Flag = 0, exactly match then
     begin
       *Nfound ← 1;
```

```
    E_point ← 2D locational code of *Nlocation;
    if E_point not in the OBSTACLE list then
      INSERT(E_point, OBSTACLE);
    return(E_point);
  end;
else if Flag = -1 then
  Bottom ← Mid - 1; /*for next loop comparison*/
else if Flag = 1 then
  Top ← Mid + 1; /*for next loop comparison*/
end;
```

The major operation of the obstacles extraction process is the collision check of a path segment with the danger nodes. Each element in the ideal direct path segment from N_{start} to N_{goal} is checked. Bresenham's algorithm for generating a path segment is based on the considerations that it avoids generating duplicate points and the use of multiplication and division operations. In addition, the algorithm can be simplified by using only integer arithmetic [Newm78].

Because the list of danger nodes is arranged in ascending sequence, a binary search method is used. The performance of a binary search is $logN_{dn}$ as described in chapter three, where N_{dn} denotes the number of danger nodes. Let N_{lp} be the number of point elements along a direct path, measured in resolution units, then the time cost of the entire collision check is $O(N_{lp}*logN_{dn})$. This result implies that the longer the path, the more point elements need to be checked and the cost is in proportion to the length of the path segment.

After collision checking, a list *SEED* is obtained. Each member in the list is used as a starting point for expansion of the obstacles region where the number of elements in the list *SEED* is bounded by N_{lp}. During the expansion process, a node at level d (root at level n, pixel at level 0) may recursively invoke the function OBSTACLE_GROWING a maximum 4^{d+1} times and a minimum of four times to locate its four main direction neighbours. In the coarser level approximation method, a node has only to be checked a maximum of four times to determine its four main direction neighbouring nodes. The expansion in each direction terminates as soon as the binary

search process locates a smaller size neighbouring node.

Dyer has shown that both the average and worst case numbers of black nodes in the quad-tree representation of a 2^n x 2^n image containing a single 2^m x 2^m region are $O(2^{m+2}-m)$ [Dyer82]. An alternative characterization of this result is that the number of nodes is $O(p+n)$ where p is the perimeter (in pixels) of the region.

Assuming that the expanded obstacle nodes are single connected, this connected region corresponds to a polygon region of size N_{lp} x N_{lp} embedded in a 2^n x 2^n grided area. Usually, more than one obstacle region may expand along a direct path and the expanded regions are not connected. Each obstacle region also corresponds to a polygon region with size 2^m x 2^m embedded in a 2^n x 2^n grided area, where $2^m < N_{lp}$. Because the obstacle regions along a direct path are bounded by N_{lp} x N_{lp} (measured in resolution units). The total number of nodes of unconnected obstacle regions is no more than a signal connected region and is of the order of its perimeter.

The result discussed above implies that a hypothetic direct path extracts the number of obstacle nodes proportional to its distance and the obstacle regions are not limited to a single connected area. The maximum times for the expansion function OBSTACLES_GROWING( ) are $O(4^{d+1} * (p+n))$.

4.5 Transformation of Navigation Space

4.5.1 Visibility Graph of Navigation Space

In the transformation stage, a visibility graph is formed which is further searched for possible flight paths from a start point to goal point. The construction of a visibility graph is the major component of flight path planning. The principle of using the visibility graph method is to obtain a path from a set of line segments connecting the initial point S to the goal point G through the vertices of obstacles [Loza79]. In a visibility graph, the nodes are the vertices of the obstacles; the edges connecting pairs

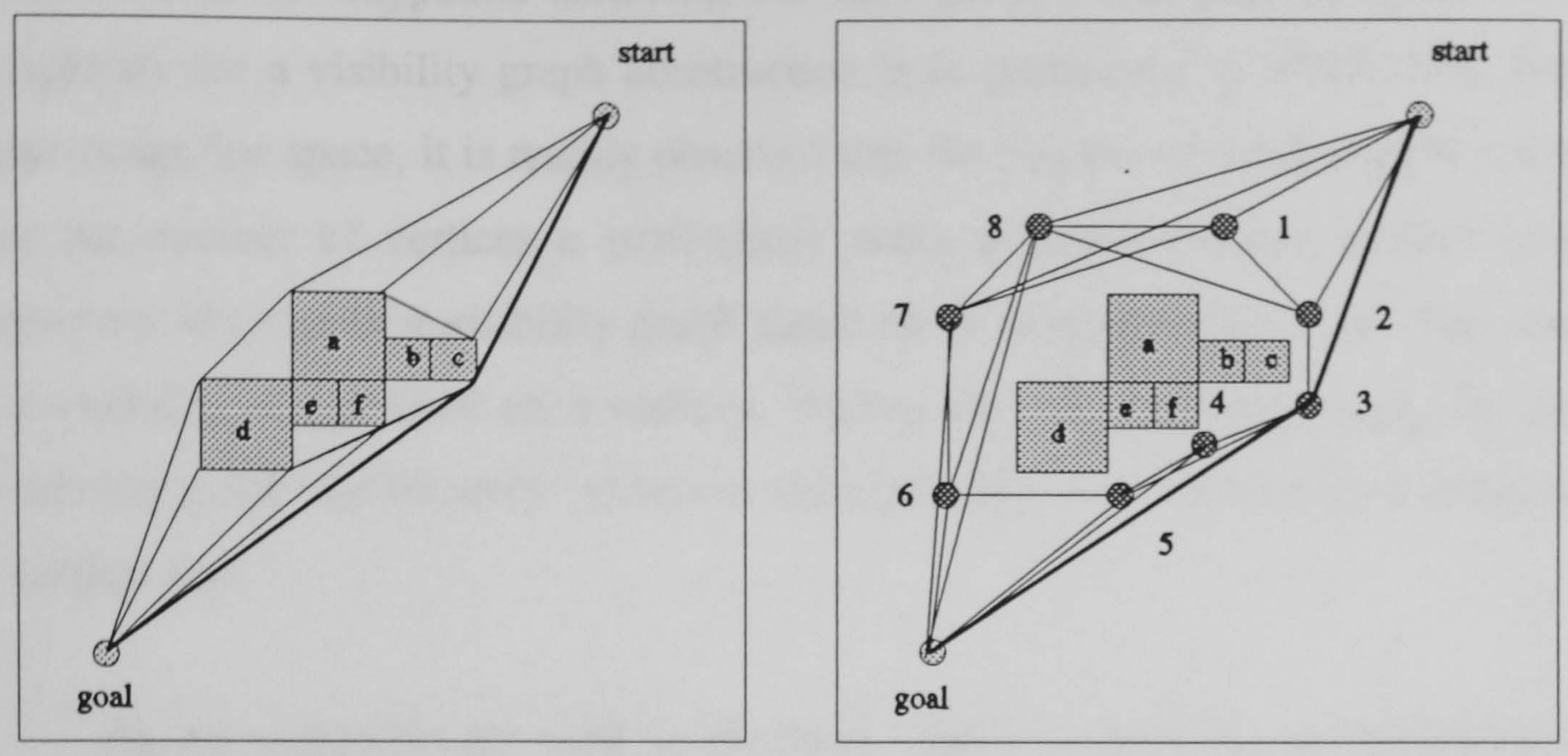
of vertices do not cross the interior of any obstacle. The resulting graph can be directly searched for a path as shown in Figure 4.10.

Before a path can be obtained, an algorithm is required for constructing a visibility graph *VG*. The construction of a visibility graph is based on the set of waypoints obtained during the acquisition stage. This set of waypoints provides the implicit geographic information of obstacle regions in navigation space. The algorithm consists of considering all pairs of points (W_{from}, W_{to}), where W_{from} and W_{to} are either start, goal, or intermediate waypoints of obstacles regions. To determine whether W_{from} and W_{to} are the endpoints of a valid flight path segment, 'collisions' are checked against obstacle regions of a path segment passing through W_{from} and W_{to}. The nodes between W_{from} and W_{to} are connected by a link in the *VG* if, and only if, there are no intersecting nodes in the open segment joining the two points.

The criteria for collision checking is similar to the one used to check for collisions of direct paths between start and goal points as described in section 4.4. A Bresenham's line generation algorithm is used to compute the point elements of a path segment formed by W_{from} and W_{to}. The collision check is performed by checking the component points along a line segment in sequence from W_{from} and W_{to} against the danger nodes list. If any point on the line is found to match a node or if it is enclosed by a node in the danger nodes list, then there is no valid path between W_{from} and W_{to}.

The test of a pair of waypoints is terminated as soon as a collision is detected, otherwise the test proceeds until W_{to} is reached. After all the possible combination of waypoint pairs are tested, the result is a visibility graph where the waypoints are the nodes of a graph the path segments formed by the waypoints are the arcs of a graph, as shown in Figure 4.10b. In Figure 4.10b, several paths exist which are composed of straight lines joining the start point to the goal point via a sequence of waypoints.

The visibility graph approach used in the transformation stage of the flight path planning algorithm is similar to the methods used in most path planning problems where



(a) VISIBILITY GRAPH FORMED BY VERTICES.

(b) VISIBILITY GRAPH FORMED BY WAYPOINTS.

Figure 4.10 The visibility graph in path searching.

a visibility graph of the obstacle space is constructed from a list of polygonal obstacles [Udup77, Loza79]. However, the following aspects are also emphasized:

Mitchell [Mitc88] has pointed out that the 'bottleneck' in the complexity of the visibility graph approach occurs in the construction of the graph. Every pair of vertices is accessed to determine whether or not the vertices are 'visible' to each other. There are $O(n^2)$ pairs, where n is the number of vertices. As described in section 4.4, the obstacle regions are obtained by gathering the obstacle nodes along a straight line path between the start and goal points and the possible obstacles are limited to those nearby the direct path (or current direction). This approach implies that the irrelevant danger nodes in the navigation space can be ignored and the number of waypoints is accordingly reduced. Because the waypoints only represent the subset of obstacles of the danger areas for an aircraft, the process in stage one provides a partial visibility graph of the total configuration space. Figure $4.9_{a\text{-}d}$ also depicts the partial configurations and their visibility graphs.

During collision checking, there are $W*(W-1)/2$ such waypoint pairs where W is the number of waypoints including the start point S and goal point G; the time complexity for a visibility graph construction is in proportion to $W*(W-1)/2$. For the same navigation space, it is readily observed that the number of waypoints W is smaller than the number of vertices n particularly when a coarse resolution level is used. Moreover, the size of a visibility graph based on W waypoints is smaller than the size of a visibility graph based on n vertices. During the path searching stage, the size of a visibility graph significantly influences the time efficiency and this issue is discussed in section 4.6.

As the waypoints are used as the detour points outside the connective obstacle regions and are generated from the hypothetic direct path, the overall distance of possible flight paths therefore tends to be short. As shown in Figure 4.9, a path is found by connecting a sequence of waypoints from the start to the goal point without departing too far from the direct path. In the next section the formal description of the transformation algorithm is given in pseudo code.

4.5.2 The Transformation Algorithm

In the transformation algorithm, the list of waypoints W_{points} is transferred into a list of records of path segments by the function SETUP(W_{points}). Each record in the list contains the information of W_{from} and W_{to} waypoints of a path segment, the distance of a path segment, and a flag used in backtracking during the path searching process (as shown in Figure 4.11).

Data item	W_{from}	W_{to}	Distance	Flag
Data type	integer	integer	integer	logic

Figure 4.11 The data structure of path segment in a visibility graph.

Function SETUP(W_{points}) checks each of the possible combination pairs of waypoints by invoking the function VISIBILITY(W_{from}, W_{to}). Bresenham's algorithm generates the point elements along the path segment formed by (W_{from}, W_{to}) for use in collision checking. If the line segment does not collide with any danger nodes, the waypoint pair, together with the distance between them (evaluated by function DISTANCE(W_{from}, W_{to})) is stored in the record. The following functions are used in the transformation stage:

```
procedure VISIBILITY(value integer W_from, W_to);
begin
  value integer Top, Mid, Bottom;
  value integer *Nfound, Flag;
  Top ← 1;
  Bottom ← number of nodes in danger nodes list;
  *Nfound ← 0;
  Bresenham(W_from, W_to);
     begin
       for each point element in a Bresenham's line segment up to, but not including W_to do
          begin
          while Top < = Bottom and *Nfound = 0 do
           begin
              Mid ← RShift (Top+Bottom,2);
              Flag ← COMPARE(element,Mid);
              if Flag = 0, collision node found then
                 *Nfound ← 1;/*no path segment*/
           end;
        end;
      return *Nfound;
    end;
end;
```

```
procedure SETUP(value integer W_points);
begin
  int Top, Mid, Bottom, COUNT;
  int *Nfound, Flag;
  Top ← 1;
  Bottom ← number of nodes in danger nodes list;
```

```
Count ← 0;
*Nfound ← 0;
for each possible combination of (W_from, W_to) and (W_from != W_to) do
   begin
     *Nfound ← VISIBILITY(W_from, W_to);
     if *Nfound = 0 then
        begin
          distance ← DISTANCE(W_from, W_to);
          save (W_from, W_to, distance, skip ← 0) as a record in an array L[COUNT];
          COUNT++;
        end;
    end;
end;
```

The collision check is based on a binary search of the elements (which constitute a path segment) in the danger nodes list. The performance of a binary search is $O(logN_{dn})$, where N_{dn} is the number of danger nodes. The time to check each collision is determined by the number of points along a path segment. The best and worst case figures are for no collision occurring at a point element after the point W_{from} and before the point W_{to}. If the average number of points checked is N_{lp} (measured in resolution units), the cost of a collision check is $O(N_{lp}*logN_{dn})$ comparisons in a binary search routine.

The time complexity for a visibility graph construction is $O(W^2)$, where W is the number of nodes in the graph. In the general case, the total cost of construction of a visibility graph from W waypoints, including the collision check, is $O(W^2*(N_{lp}*logN_{dn}))$. This approach not only allows a digital model to adopt the path planning method of a geometry model, but also improves the time complexity by building a partial visibility graph of the path searching space and thereby reduces the number of vertices.

4.6 Flight Path Searching Stage

4.6.1 Path Searching Approaches

Within the range of possible flight paths, there is likely to be one which optimises some specific operational requirement (for example, direct operating costs for a civil aircraft or time to height for a military aircraft). At the same time, the flight path must satisfy the constraints of operation of an aircraft which is governed by the laws of physics, air traffic regulation, situation of the gaming area and flight safety considerations. There will also be other criteria which influence ease of solution or operation, without significantly compromising the optimum solution.

It is important to understand the difference between finding an 'optimal' solution and finding a 'good' solution. The difference lies in the fact that finding an optimal solution often entails an exhaustive search because it may be the only method to determine whether or not the best solution has been found [Rich86]. Finding a good solution means finding one that conforms to a set of constraints regardless of whether or not a better solution exists.

Dynamic programming [Whit69] is frequently used to solve optimization problems and is commonly regarded as too cumbersome for real time use. When it is employed in real time, overly simplified models are often used [Chan85]. An optimal path may not be appropriate to real-time airborne requirements because of the possible degradation in performance to extract the path.

For a large number of nodes in a visibility graph, the computation load of the exhaustive approach is prohibitive, and more efficient heuristic based search methods are used. For example, for a map given in the form of a list of polygonal obstacles, the shortest path can be searched in the corresponding visibility graph using Dijkstra's algorithm [Dijk59], which is also known as the A^* algorithm [Nils80, Rich86]. The A^* algorithm is given in the Appendix one and will be implemented in the next chapter.

The following points are major considerations in the proposed real-time flight path planning algorithm :

- Firstly, the time to search for a solution with the minimum effort subject to real-time requirements. For example, the length of a solution path and the actual number of nodes traversed influences the speed of a search.

- Secondly, the 'quality' of the path solution with respect to minimum distance and time to fly the selected path.

The visibility graph is represented in the form of a list of flight path segments, as described in the last section. The path planning problem has been transformed into a discrete problem of searching the visibility graph between a start node and a goal node. The following conventions and notations are used in searching a visibility graph:

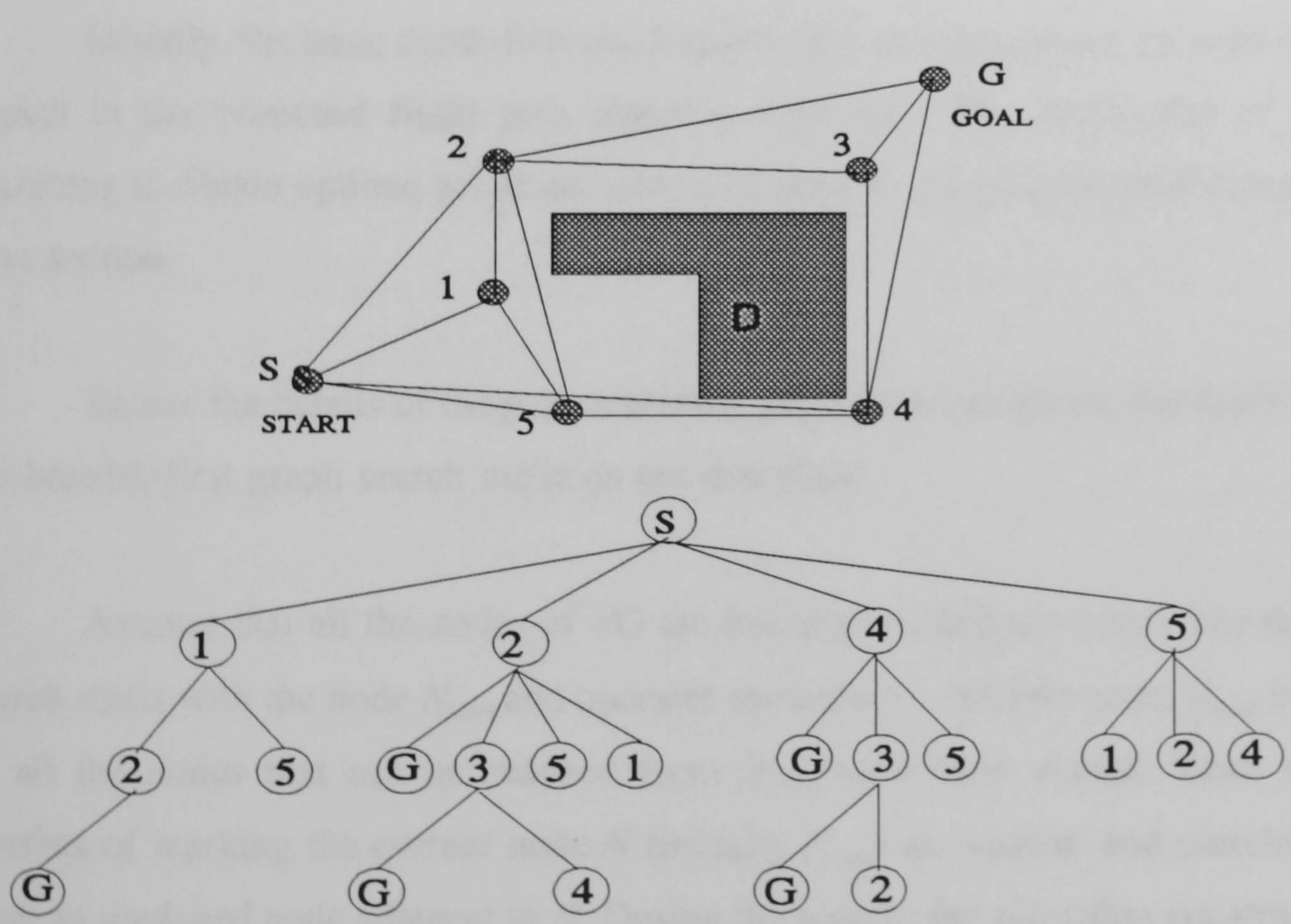


Figure 4.12 A visibility graph and its tree structure.

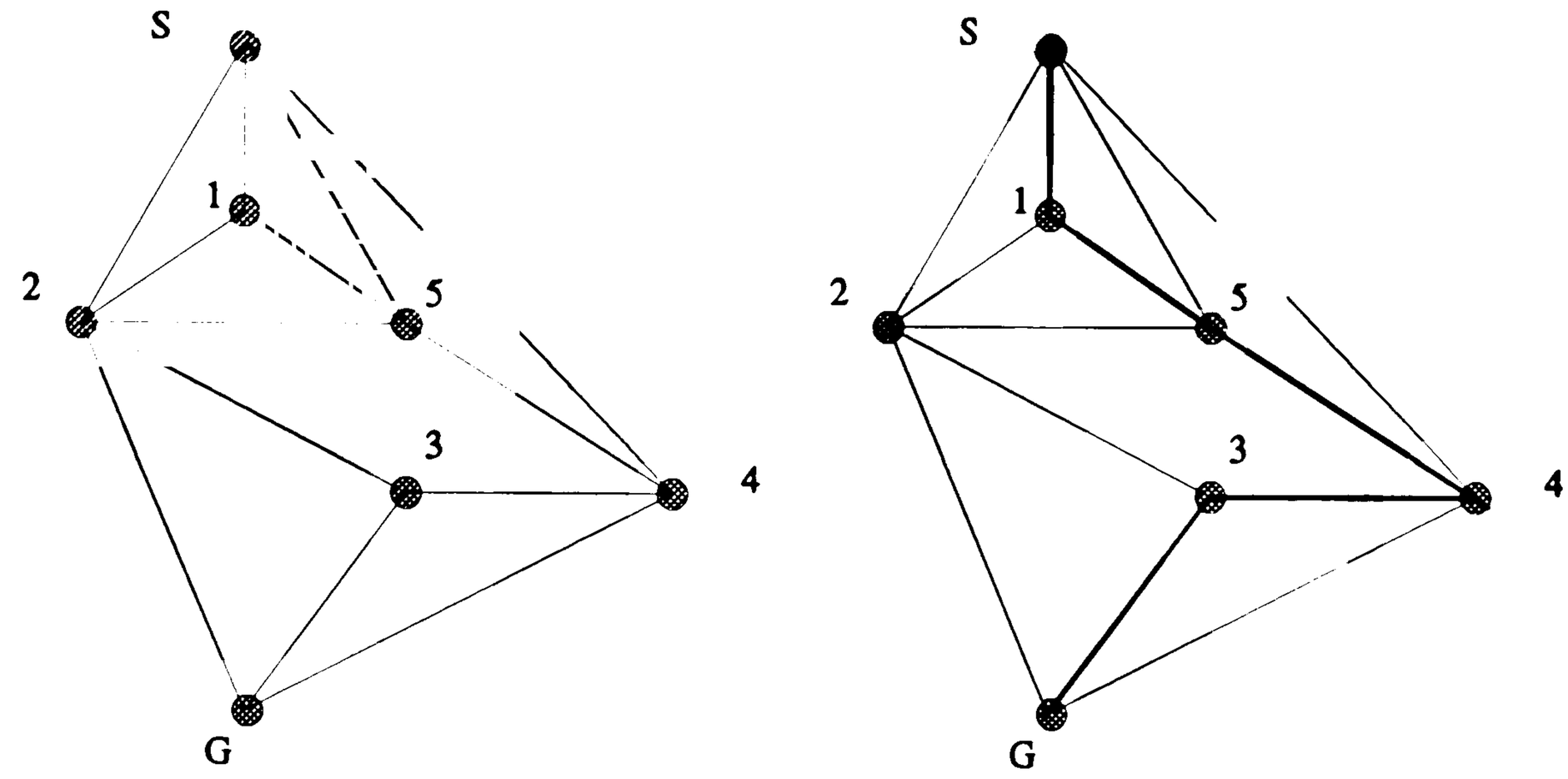
Let $VG = (W_{points}, A)$ be an undirected graph consisting of a set of nodes W_{points} and a set of arcs A. Let W be the number of nodes and r be the number of arcs in the VG. N_{start} and N_{goal} represent the start and goal nodes in the visibility graph VG, respectively. N' and N represent two adjacent nodes connected by an arc. In the worst case, $r \in O(W^2)$. For example, Figure 4.12a shows a visibility graph representing path segments of a navigation space. There are seven nodes in the graph including the start and goal points. It is redrawn as a tree structure in Figure 4.12b where the goal node G and other nodes appear more than once in the tree to illustrate its construction.

Several techniques are used for searching for a possible solution in a graph, including depth-first, breadth-first and heuristic search [Aho74, Rich86]. The depth-first and breadth-first search are the most commonly used methods for searching a path in a visibility graph VG. The hill-climbing and least cost methods [Rich86] are also examples of the heuristic search, based on depth-first searching and they are normally used to minimize or maximize some aspect of constraints such as path distance or the number of connection points along a flight path.

Initially, the basic depth-first and breadth-first search methods are applied to find a path in the proposed flight path planning algorithm. The application of heuristic searching to obtain optimal solutions subject to specific constraints are discussed in the next section.

Before the details of the path searching algorithms are given, the depth-first and the breadth-first graph search methods are described:

Assume that all the nodes of VG are initially marked *unvisited*. The depth-first search starts with the node N_{start} and operates recursively, until the node N_{goal} is reached or all the nodes that can be reached from N_{start} have been visited. Each recursion consists of marking the current node N (initially N_{start}) as 'visited' and searching a VG from an *unvisited* node adjacent to N. During the search, the algorithm constructs a tree T as follows. Whenever a node N is considered, which leads to an *unvisited* node N', N' is added to T together with a path segment connecting N' to N. T is the spanning tree



(a) GRAPH REPRESENTATION.

(b) SPANNING TREE OF (a). IN BOLD LINE.

Figure 4.13. Path searching in a visibility graph.

[Aho74] of the subset of *VG* visited so far. Figure 4.13 shows a visibility graph and one of its depth first spanning trees. Beginning from S, the sequence of vertices visited is S → 1 → 5 → 4 → 3 → G. If N_{goal} is ultimately attained, a path is constructed by tracing the arc in *T* from N_{goal} to N_{init}. The method has a time performance $O(r)$, where *r* is the number of path segments. As each node is marked as *visited* the first time it is visited ensuring that each path segment is traversed only once.

A depth-first method is often easy to implement and can produce a good solution very quickly. However, the depth-first method is inefficient in situations where particularly long branches of a graph are explored only to find that there is no solution. In this case, the depth-first method will waste considerable time, not only in exploring this branch, but also in backtracking to the goal. A depth-first search is certain to find the goal because, in the worst case, it degenerates to an exhaustive search.

Another method applied for searching the *VG* is the breadth-first search method [Aho74]. Breadth-first searching is the opposite of depth-first searching. This method uses a queue of nodes denoted by OPEN which is initialized as a queue containing N_{start} only. All the nodes of the *VG* are initially marked *unvisited*. Breadth-first search works iteratively. While OPEN is not empty, it considers the first node *N* in OPEN and removes it from OPEN. If any of the nodes adjacent to *N* is N_{goal}, it terminates. Otherwise, it marks every *unvisited* node *N'* adjacent to *N* as *visited* and places *N'* at the end of OPEN.

During the search, a spanning tree *T* of the subset of *VG* visited so far is constructed as in the depth-first search method. If N_{goal} is ultimately reached, a path is constructed by tracing the arcs in *T* from N_{goal} to N_{start}. The breadth-first method takes $O(r)$ time. Every node visited is placed in OPEN once, so that the process is executed iteratively once for each node and each arc is examined once only.

A breadth-first search is guaranteed to find a solution if one exists [Rich86]. A major disadvantage to breadth-first searching is that substantial effort is expended to find a goal which is located several levels deep within the graph. It is particularly inappropriate in situations where there are many paths that lead to solutions but each of them is quite long. In such situations, the depth-first search is more effective.

Both the depth-first and breadth-first search methods have been applied to solving the path search problems in this thesis. The formal description of the path planning algorithms and the implementation of search methods discussed above will be described in detail in the following section.

4.6.2 Path Searching Algorithms

Before the actual algorithm to find a path between N_{start} and N_{goal} is described, several support functions are needed. First, a routine is needed to determine if there is a valid path segment between two waypoints. The function MATCH(from,to) returns

either 0, if no such path segment exists, or the distance between the two waypoints, if there is a valid path segment:

```
procedure MATCH(from,to);
value integer n;
value integer from,to;
value integer N,N'; /* waypoints of any path segment L[n] */
begin
  while not the end of path segment list L[n] do
    begin
      if (from == N and to == N') connection is found
          return the distance in record L[n];
    end;
  return 0;
end;
```

Another necessary routine is FIND(N,N'). When a waypoint N is given, FIND(N,N') searches the list of flight path segments for a connecting path segment. If FIND(N,N') detects a connecting path segment, it returns the locational code of the destination waypoint and its distance; if it does not find one, it returns 0. The FIND( ) routine is as follows:

```
procedure FIND(N,N');
value integer N,N';
value integer n;
begin
  n ← 0;
  while (n < r, the final record of L[r] is not reach) do
    begin
      if (N == start value of L[n]) and (the skip flag of L[n] == false)
        begin
          N' ← goal value of L[n];
          skip flag of L[n] ← true;
          return the distance in record L[n];
        end;
      n ← n+1;
    end;
  return 0;
end;
```

A path segment that has set the **skip** field (flag) to TRUE is not a valid connection. If a connecting path segment is found, it marks the connection's **skip** field as active (visited) and this process is used to control backtracking from 'dead ends'. The recursive backtracking is accomplished by using a backtrack stack. The backtracking operations are stack-like operations (first-in, last-out). Thus, as the search explores a path, it pushes path segments onto the stack as it encounters them. At each dead end, the search pops the last path segment off the stack and tries a new path from that waypoint. This process continues until either the search reaches the goal or exhausts all path segments.

A depth-first search explores each possible path to its goal before trying another path. To determine a route from the start point to goal point requires the use of a records list $L[r]$ (r is the number of records in the list) that contains the information about existing path segments between any pair of waypoints. Each record in the list contains the information of the 'from' and 'to' waypoints of a path segment, the distance of a path segment, and a flag field **skip** for backtracking (initially set to 0). Those path segments are finally linked together to form a flight path from start to goal point. The depth-first search routine works as follows :

(1) The function MATCH checks the path segments list to see if there is a direct flight path between the N_{start} and N_{goal} points. If there is, then the MATCH routine has reached the N_{goal}; the routine then pushes this connection onto the stack for output.

(2) If there is no path, function FIND checks to see if there is a connection between N_{start} and any waypoint. If there is, the FIND pushes this connection onto a stack and calls the path planning routine recursively. In this case the waypoint becomes the temporary start point during the current stage.

(3) If there is no connecting path, backtracking takes place. For example, if the function FIND encounters a dead end, it simply back tracks to an earlier point

in the process and tries another approach from that waypoint. Backtracking is applied by using a recursive routine which removes the previous node from the stack and calls the path planning routine. This process continues until either the routine reaches the goal point or exhausts all path segments. A **skip** field is necessary in backtracking to avoid visiting the same connection over and over again.

(4) The status of the stack contains the route between N_{start} and N_{goal} point and determines the success or failure of the path planning routine. If a set of path segments is found, then the stack holds the solution. Otherwise, the stack is empty implying that no path has been found.

Figure 4.14a shows a simple example of this traversal method. A tree representation of a visibility graph is used to illustrate the order in which the graph nodes are visited. The traversal sequence of finding a path in the tree depends on the

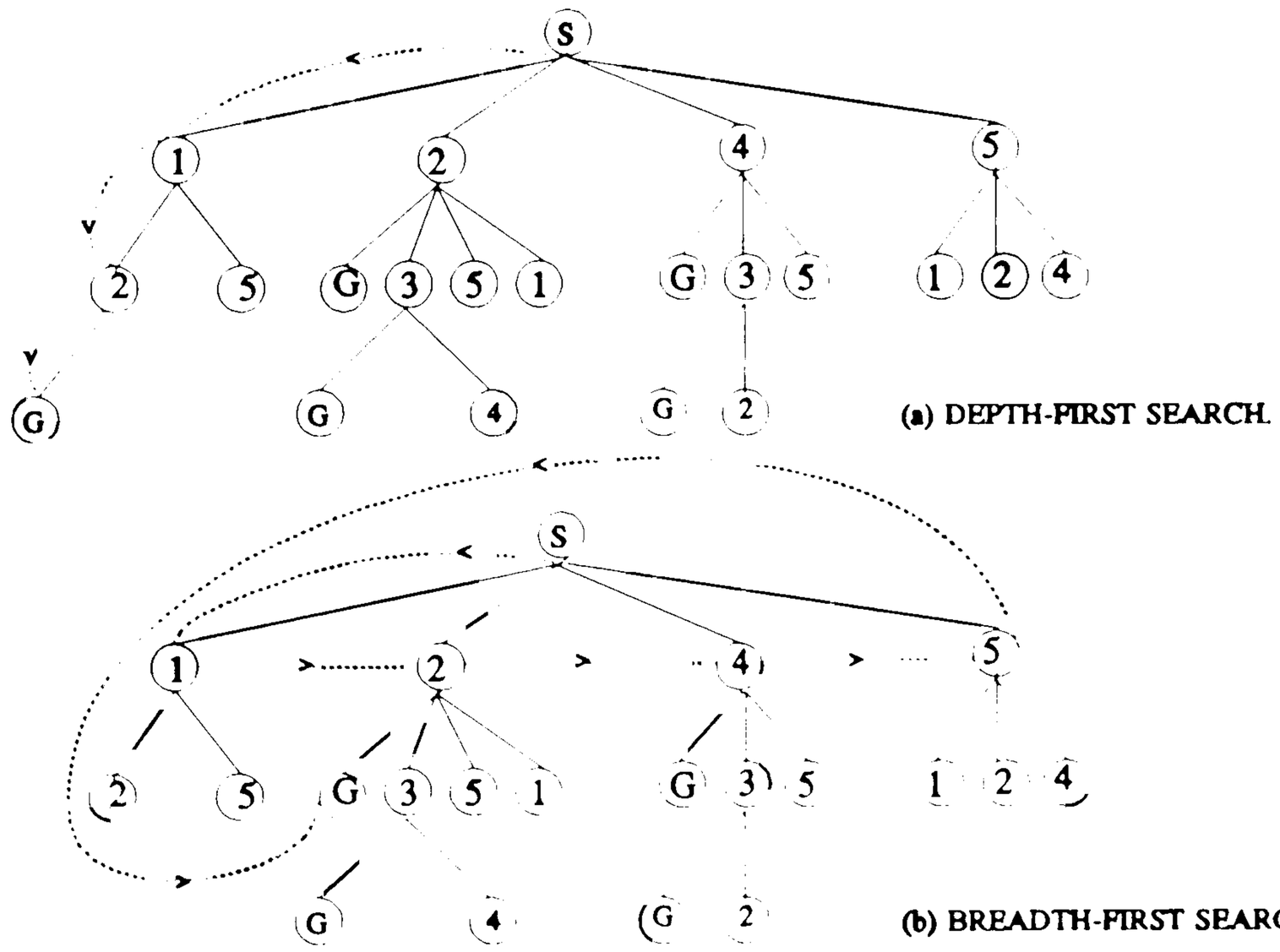


Figure 4.14 Depth-first and breadth-first traversal of a visibility graph.

physical structure of the tree; some possible sequence are as follows:

$S \rightarrow 2 \rightarrow G$,

$S \rightarrow 1 \rightarrow 2 \rightarrow G$,

$S \rightarrow 4 \rightarrow 3 \rightarrow G$.

The formal depth-first search algorithm DEPTH_SEARCH( ) is shown below:

```
procedure DEPTH_SEARCH(from,to);
value integer from,to;
value integer n;
begin
  if MATCH(from,to) != 0 a connection path is matched do
     begin
        push the connection (from,to,distance) onto the stack;
        return;
     end;
  if FIND(from,any waypoints) != 0, a connected path segment is found do
     begin
        push the connection (from,to,distance) onto the stack;
        DEPTH_SEARCH(any waypoint, to);
     end;
  else if no path segment found and not empty stack do
     begin
        pop the previous connection (from,to, distance) from the stack;
        DEPTH_SEARCH(from, to);
     end;
end;
```

The depth-first method has a time performance $O(r)$, where r is the number of path segments. Each time the function FIND(from,any waypoints) is invoked, the **skip** field of each path segment is marked *visited* the first time it is visited and each path segment is traversed only once.

The breadth-first search checks each node on the same level of a graph before it proceeds to the next deeper level. The algorithm checks all waypoints that are connected to the start point to see if they also connect with the destination N_{goal}. If there

is no path, function FIND checks to see if there is a connection between any waypoints at the next deeper level and the start point N_{start}. If there is, the procedure FIND pushes this connection onto a stack and calls the path planning routine recursively. In this case the waypoint becomes the temporary start point, during this current stage. The other processes are the same as the depth-first method. Figure 4.14b illustrates this traversal method with G as the goal and shows that the search visits the waypoint in the following sequence to find a first path from start to goal:

S → (1, 2, 4, 5), 1 → (2, 5), 2 → G.

The depth-first search method is modified to the breadth-first search as follows:

```
procedure BREADTH_SEARCH(from,to);
value integer from,to,any waypoint;
value integer distance;
begin
  while FIND(from,any waypoint) != 0, a connected path segment is found do
    begin
     if MATCH(any waypoint,to) != 0 a connection is matched do
       begin
         push the connection (from,to,distance) onto the stack;
         push the connection (any waypoint,to,distance) onto the stack;
         return;
       end;
    end;
if FIND(from,any waypoints) != 0, a connected path segment is found do
    begin
       push the connection (from,to,distance) onto the stack;
       BREADTH_SEARCH(any waypoint, to);
    end;
else if no path segment found and not empty stack do
      begin
         pop the previous connection (from,to, distance) from the stack;
         BREADTH_SEARCH(from, to);
      end;
end;
```

The breadth-first method checks all path segments (that the temporary start waypoint connected) to see if they are connected to the destination waypoint. Every path segment is visited by the function FIND(from, any waypoint) and has its **skip** field marked, so that the process is executed iteratively once for each path segment. Each path segment is examined once only. The breadth-first method also takes $O(r)$ time.

4.7 Applying Constraints and Heuristics

In the flight path planning algorithm, constraints can be applied in the search for an optimal flight path either to minimize the number of waypoints which form the path from start to goal, or to minimize the distance of the flight path. Cost functions are defined to evaluate the effect of adding waypoints to determine which set of waypoints should be expanded in the path planning routine. This evaluation function could be calculated for any path segment in the search space. The constraints are also applied to heuristic search methods in the flight path planning algorithm.

The depth-first method search, which relies on moving from one waypoint to another can be improved by the addition of heuristics [Nils80, Rich86]. Heuristics are rules that qualify the possibility that a search is proceeding in the correct direction and the implementation of heuristics to flight path searching is achieved by including the knowledge of constraints in the routine that searches for a connecting path segment. For example, the search algorithm that minimizes the number of connections will use heuristics based on the assumption that the longer the distance covered, the greater is the possibility that the next waypoint will be placed closer to the destination, thereby reducing the number of connections. This type of search method is also known as hill-climbing [Rich86].

The heuristic search method and the constraint of minimising the number of connections has been incorporated in the flight path planning algorithm to find a satisfactory solution. During the search, a path segment is chosen that is as distant as possible from the current position in the hope that it will be closer to the destination. A modified FIND( ) routine is used to search the entire path segment list in order to

find the connection that is farthest away from the start point at each search stage. The modified routine FIND_MINI_CONNECT( ) is as follows:

```
procedure FIND_MINI_CONNECT(N,N');
value integer N,N';
value integer n,current;
value integer distance;
begin
  n ← 0;
  distance ← 0;
  while not reaching to the last but one record of L[r] do
    begin
      if (N == start value of L[current]) and (the skip flag of L[current] == 0)
        begin
          if ( distance value of L[current] > distance)
            begin
              n ← current;
              distance ← distance value of L[current];
            end;
        end;
    end;
  if farthest connection L[n] is found
    begin
      N' ← goal value of L[n];
      skip flag of L[n] ← 1;
      return the distance in record L[n];
    end;
  return 0;
end;
```

On the other hand, the method of minimizing the length of the flight path entails an opposite approach to minimise the connections. This type of search method is called a least-cost search [Rich86]. If the flight path distance is taken as the cost function, in the routine FIND( ), the program will take the shortest connecting path in all cases so that the route found has a better chance of having the shortest distance. The function based on minimising the distance of the flight path is as follows:

```
procedure FIND_MINI_DIST(N,N');
value integer N,N';
value integer n,current;
value integer distance;
begin
  n ← 0;
  distance ← any value larger than the side length of the gaming area;
  while not reaching to the last but one record of L[r] do
  begin
    if (N == start value of L[current]) and (the skip flag of L[current] == 0)
      begin
        if ( distance value of L[current] < distance)
          begin
            n ← current;
            distance ← distance value of L[current];
          end;
      end;
  end;
if farthest connection L[n] is found
  begin
    N' ← goal value of L[n];
    skip flag of L[n] ← 1;
     return the distance in record L[n];
   end;
  return 0;
end;
```

While both the depth-first and breadth-first search techniques are concerned with finding a path, the initial solution may not a good one, for example, the result is not a shortest path or it has too many connections before the goal is reached. The results depend on the physical organization of the graph as described in section 4.6.1. Two methods were used to improve the likelihood of finding a 'good' and 'preferably optimal' solution.

The choice of heuristic searching methods depends on the constraints to be minimized or maximized. Heuristic techniques tend to work better than blind searching [Rich86] and the implementation of heuristics to flight path searching has shown that

generalized path searching can be applied to the proposed terrain representation and the modelling of navigation space.

4.8 Summary of Flight Path Planning

In this chapter, a terrain oct-tree based flight path planning algorithm has been discussed. Unlike many other motion planning approaches using predefined obstacle models, the obstacles need to be explored in the navigation space. In terms of motion planning problems, the proposed terrain oct-tree representation is categorized as a cell decomposition approach; while most cell decomposition terrain models adopt a connectivity graph of free space for path searching, instead of using such a time consuming search method, the visibility graph approach is adopted to gain the benefits of space and time efficiency.

A set of waypoints outside the obstacle regions is obtained by identifying the obstacles and locating the diagonal neighbours of each vertices of the obstacles. After a collision check procedure has been applied to each pair of waypoints, a visibility graph is constructed. The generalized graph searching methods and heuristics are applied to path searching at this stage, to generate an acceptable global flight path.

The features of the flight path algorithms are the following:

- A terrain oct-tree is the only input data, no other terrain features data are used and the space requirement is minimized.
- The obstacles are not predefined and can be extracted from the terrain oct-tree as the navigation conditions vary. This feature makes the algorithm amendable to a real-time navigation environment.
- The visibility graph constructed is a partial visibility graph of the total configuration space, where the construction time is reduced to a minimum.
- By exploiting the hierarchical features of a quad-tree, the number of the waypoints is 'controlled' by the resolution of the boundary nodes, thus the size of a visibility graph can be reduced.

- The projection codes allow the path planning process to be performed in a two-dimensional space. The use of terrain oct-tree avoids the expensive cost of path planning in three-dimensional space.

Observation of the time performance is performed by executing the program and evaluating the characteristics of each stage of the algorithm. The measurements include the time to generate the waypoints, the time to construct a visibility graph and the time to find a path. The results of time performance have been used to 'guide' a real-time dynamic flight path planning algorithm, which is described in chapter 5 and 6.

In the next chapter, a real-time dynamic flight path planning algorithm is developed to allow an aircraft to continuously update its flight path 'during the flight' in response to mission requirements and changing obstacles. The constraints on flight path planning and the heuristic search are also adopted to the methods developed in this chapter.

CHAPTER 5

IMPLEMENTATION OF FLIGHT PATH PLANNING ALGORITHMS IN A REAL-TIME DYNAMIC ENVIRONMENT

5.1 Introduction

Traditional flight mission planning by aircrews consists of paper charts, target photographs and manual plotting to measure distances and angles. The method is slow, tedious and error prone. Moreover, a detailed mission plan for a military application may take tens of man hours to prepare and to check. Furthermore, initialization of avionic systems with mission-specific data can be time consuming and inaccurate, which unduly increases pilot workload, decreases reaction time, reduces mission effectiveness, and can result in additional fuel requirements.

With recent advances in computer performance and mass storage technology, a mission planning system allows a pilot to perform all the planning requirements on a ground-based workstation prior to takeoff. A mission planning system display maps, photographs and terrain data that are precisely geographically aligned. Distance, heading and fuel are automatically computed after the pilot selects the flight path which defines navigation turnpoints, offset points, the start point and the goal point.

In a military system, with practically unlimited ground-based computing resources, sophisticated algorithms are used to generate combat mission data including colour maps, radar predictions, 3-D views and flight forms. This digital data is subsequently up-loaded prior to the mission to initialize the aircraft computers and to record in-flight maintenance and operational data. The entire process to plan and produce a mission plan is typically completed 20-30 minutes [Fair91] before take-off.

During the mission, however, new situations may occur or new information may become available through sensors which require the aircraft to deviate from the pre-planned flight path, for example, 'pop-up' threats, a new destination or previously

unknown terrain obstacles. In these cases, the remainder of the preplanned path may no longer be relevant and a real-time path generation scheme is required to direct the aircraft to the designated target based on the current mission situation.

In this chapter the flight path planning algorithm proposed in chapter four is applied to a real-time dynamic environment for aircraft manoeuvring during a flight mission. The statistical results derived from time performance analysis are used as a guideline for designing a real-time dynamic algorithm.

5.2 Real-Time Path Generation

5.2.1 Requirements

Based on the assumption that the navigation space (obstacles, terrain, threats) is static and is fully known a priori, flight mission planning or route planning can be completed before commencement of the mission. However, if a new path is requested during a flight mission, the path planning process is performed in the real-time airborne environment and this operation may be repeated several times throughout the mission.

The replanning of a new flight path should be completed within a few seconds to meet real-time navigation requirements. This time interval includes the flight path planning computation (described in the last chapter) and also the determination of the path searching space. For instance, it may happen that there is no path to reach to the goal under the current flight conditions, and appropriate actions must be taken, which could include extraction of an alternative search space based on a different safety altitude or changing to a different resolution level of the terrain oct-tree.

Real-time dynamic path planning contains the following phases:

- During a predefined flight mission, a change of flight course is given, often in the form of a diversion to a new goal point or a change of minimum safety altitude. Assuming that the flight path planning algorithm takes T seconds to

replan a new route, the aircraft maintains its current flight path for T seconds before the new flight path is initiated.

- Before the replanning process can be commenced, the first problem is the selection of the new start point. The new start point could be defined as the aircraft position in T seconds time relative to the current path. This situation implies that the flight path planning algorithm has the ability to estimate the complexity of the new obstacle space.

- During this period of T seconds, the algorithm performs the extraction of obstacles, the construction of the visibility graph and path searching. Moreover, if a new path is not found, then the algorithm can propose a new path by changing the minimum safety altitude.

The flight path planning algorithm described in chapter four is performed on a terrain oct-tree representing a navigation space without using any other data structure of features (linear or polygonal features). This is based on the fact that the number of terrain objects which must be treated as obstacles tends to decrease as the flight altitude increases; by giving a flight altitude as a constraint, the obstacles can be easily abstracted from a terrain oct-tree. Since the obstacles change with respect to time as the flight altitude changes, a static polygonal representation of obstacles is not suitable for a dynamic airborne environment. In contrast to polygonal representations, the elevation data gives a flexible way to locate terrain obstacles.

However, auxiliary obstacles information and data specifying features can always be superimposed to the navigation space. For example, a predefined forbidden region in a navigation space can be transformed into a set of nodes of a corresponding quad-tree representation [Mark85, Same80, Garg82] for collision checking during the planning process.

5.2.2 Pre-processing Schemes

Several approaches have been adopted to real-time aircraft flight path generation algorithms [Chan85, Lau87, Meng87]. Pre-processing of the search space to reduce the computation cost is commonly used in the real-time dynamic environment. Each approach highlights different points according to the objectives of the applications.

Lau's flight paths are composed of a 'local path' and a 'Route Plan Path' in two parts [Lau87]. The Route Plan Path constructs a flight corridor between every pair of waypoints and divides each corridor into a coarse grid. Each cell is assigned a cost based on the elevation of its highest point, threat exposure, and deviation from a direct route. A dynamic programming method [Whit69] yields the best path and selects a path to the end of the mission. The Route Plan Path computations are completed before the flight begins. The local path segment is generated in real-time by using the cost defined in the Route Plan Path stage and 'runs' a short distance (about 30 seconds of flight) 'in front' of the aircraft for terrain avoidance requirements.

Meng presents a real-time route planner for robotic air vehicles by modelling the free space skeleton based on a Voronoi graph in searching for flight paths [Meng87]. In order to reduce the real-time cost to a minimum, the Voronoi graph is computed for a given set of objects described by their geometrical property (closed boundary). This graph is computed and interpreted before the flight mission.

In Meng's approach, the three-dimensional space is 'sliced', starting from the base altitude, in vertical intervals of altitude. A set of Voronoi graphs is obtained corresponding to each altitude layer. The algorithm searches for a path either within a contour slice for the chosen altitude or it may switch between layers when no safe path can be found for the current altitude. The real-time computation cost to search the graph for the path is only the cost of computing the retractions of the start and goal points and path searching of the Voronoi graph.

Chan focuses on path optimization [Chan85] in developing a real-time algorithm which achieves an optimal solution by means of the storage of a massive pre-computed data base of intermediate results using dynamic programming. In order to reduce the real-time computation overhead, the bulk of numerical processing of the cost index in a dynamic programming operation is pre-computed and stored for on board access. The optimal cost database stores the cost factors for every grid location of a gaming area. A number of additional database including DTED, SAM threat are also stored on-board. The database storage requirement of a 100 km by 500 km region is approximately 110M bits.

The new path generation involves updating the intermediate cost only in the area covered by the aircraft sensors where new threats or obstacles are found. A coarser path is obtained by applying dynamic programming on a threats masked grid file with large column and row spacings. The real-time computation cost is then reduced by limiting the numerical processing of dynamic programming associated to this coarser path. A parallel processing system is proposed to further reduce the computation time.

After the optimal costs data base has been updated, the flight path is taken from the heading and flight path angle data base, starting from the current position to the goal position where the accumulated local cost is minimal. Because all the numerical processing is pre-computed, the search of the graph proceeds in the direction of flight and the time cost is minimised.

Both Lau's and Chan's approach are grids methods to represent navigation space, with cost values embedded in the database. The application of a coarser path is analogous to the cell decomposition approach (described in the chapter two). Dynamic programming is traditionally considered too time consuming for real-time applications at fine scales. Even with a small scale (coarser path) area of cover, dynamic programming still requires substantial numerical processing. Although the pre-computing process significantly reduces the computation costs, the costs are still prohibitive.

The different approaches discussed above have a common feature that pre-processing schemes are applied to provide real-time flight path generation. These pre-processing tasks can be classified in two categories, one of which pre-defines the graph of the search space according to the terrain altitude and the other which pre-computes the cost values embedded in a graph of the search space.

Pre-definition of a search graph is used to model the search space (such as the visibility graph or Voronoi graph) and influences the efficiency of the searching process, whereas, the use of pre-computed cost indices concentrates on searching methods (such as dynamic programming and A* search) and the quality of the solution.

5.2.3 The Searching Space

As discussed in the previous section, most prior flight path planning algorithms focus on the problems in the path searching stage which comprise pre-processing of the cost value, optimization and searching methods. The threats and prohibitive areas are predefined in the most cases. During the flight, any pop-up threats, uncharted terrain or previously unreported obstacles are assumed to be detected by on board sensors and mission replanning is kept to a minimum to match the real-time requirement.

If a path searching process is directly performed on a grid model, a cost value (in terms of distance, threat, elevation, fuel or heading etc.,) is attached to each grid point corresponding to the row and column spacings; the aircraft is assumed to move between points on the grid. Therefore, with the exception of boundary points, there are eight possible directions of motion from any points, as shown in Figure 5.1. A grid graph is then considered where the nodes are the grid points and the edges connect adjacent grid points. This implies that the grid graph is at most n nodes and a maximum of $8n$ edges which is cumbersome for real-time path searching.

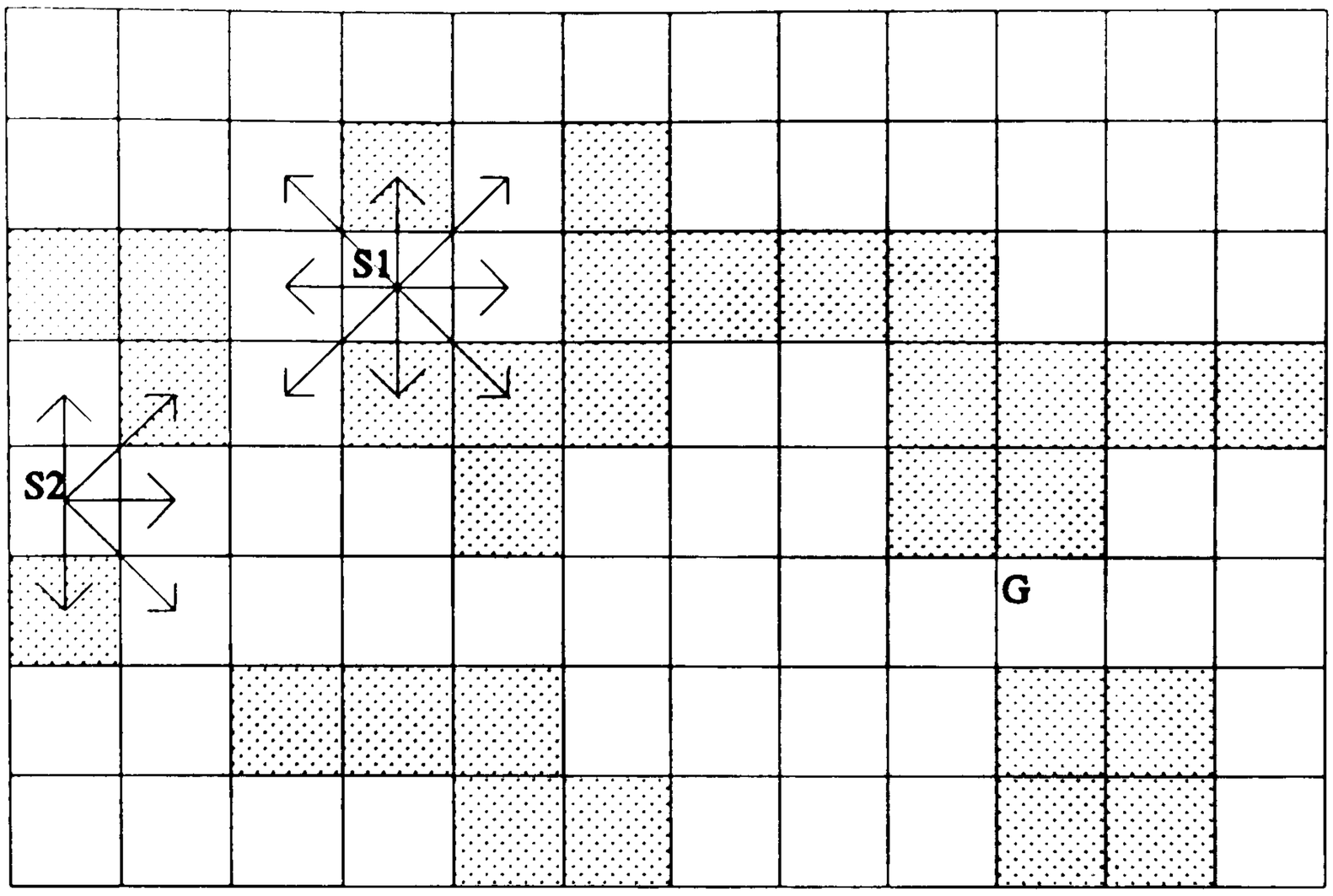


Figure 5.1 Each non-boundary cell may has most 8 possible directions of travel.

The flight path planning algorithm described in chapter four uses a visibility graph for the search space. A collision avoidance approach is applied to find a flight path under constraints of flight altitude, flight distance or the number of waypoints. Some other cost factors (threat, fuel or heading etc.,) can also be superimposed on a visibility graph.

Because the size of a visibility graph is proportioned to the number of vertices of obstacles and the number of vertices is always less than the number of grid point, the size of a visibility graph is smaller than the size of a grid graph. Assuming both the visibility graph and the grid graph of a navigation space are searched for shortest paths, the visibility graph approach is efficient in comparison with the grid approach, especially when a partial visibility graph is used.

5.2.4 Optimal Solutions

Many flight path planning algorithms make use of the cost knowledge of the navigation space to develop an optimal path [Lizz85, Chan85, Lau87], with the intention of minimizing some objective function which specifies the cost of motion along the path. Depending on the terrain modelling method and the optimization criteria, this results in various versions of the shortest path problem.

It is shown in Chan's approach that finding the truly *optimal path* by means of dynamic programming techniques would require a costly exhaustive search, but without hardware and pre-processing support, it is inappropriate for a real-time airborne environment.

Heuristic methods are defined as the application of task-dependent knowledge to minimize the size of the search space. The A* algorithm is a heuristic approach that attempts to combine the benefit of depth-first and breadth-first searches by guiding the direction of the search pattern using heuristics. It has been proven [Nils80] that the A* algorithm is admissible*. That is, it always finds an optimal path if one exists.

The visibility graph can be searched for minimum-cost paths by applying the A* algorithm [Mitc84, Kamb86]. In the method a visibility graph is iteratively visited by following paths originating from N_{start}. At the beginning of each iteration, there are some nodes that the algorithm has already visited, and there may be others that are still unvisited. For each visited node N, the previous iterations have produced one or several paths connecting N_{start} to N, but the algorithm only needs to store a list of the start and end points of each path segment of a path of minimum cost among those so far constructed.

---

* A search algorithm that is guaranteed to find an optimal path to a goal, if one exists, is called admissible [Nils 80].

At any instant, the set of all such paths forms a spanning tree T of the subset of a visibility graph so far explored. T is represented by associating with each visited node N, a pointer to its parent node in the current spanning tree T.

A cost function $f(N)$ is assigned to every node N in the current spanning tree. This function is an estimate of the shortest distance path in the visibility graph connecting N_{start} to N_{goal} and constrained to pan through N. The cost function is computed as follows:

$$f(N) = g(N) + h(N)$$

where $g(N)$ is the cost of the path segment between N_{start} to N in the current spanning tree T and $h(N)$ is a heuristic estimate of the cost $h^*(N)$ of the minimum-cost path between N and N_{goal} in a visibility graph.

It has been shown [Fred84] that the worst-case running time of the A* algorithm is $O(r + nlogn)$ for a graph with r edges and n nodes. In contrast to a grid graph, a visibility graph reduces the size of the search space, because the values r and n in the visibility graph are always smaller than the values r and n of a grid graph. Generally the time performance in a visibility graph is considerable better than a grid graph.

5.2.5 Summary of Real-time Path Generation

Flight path planning is required to be performed repeatedly in real-time throughout a flight mission and this requirement implies substantial information processing demands in a confined airborne environment. For instance, an algorithm must maintain and update a model of navigation environment together with a set of flight plans that will minimize the cost and risk during the flight.

The algorithm proposed in chapter four concentrates on modelling of the searching space and reducing the searching space before the search process is performed. The subsequent planning process is thus performed on a smaller visibility graph to improve the time performance.

As it is anticipated that an aircraft flight path will need to be modified in real-time to match flight conditions and changing obstacles in the environment, the terrain database is accessed for every flight plan change in order to re-form the visibility graph, based on the current obstacle space.

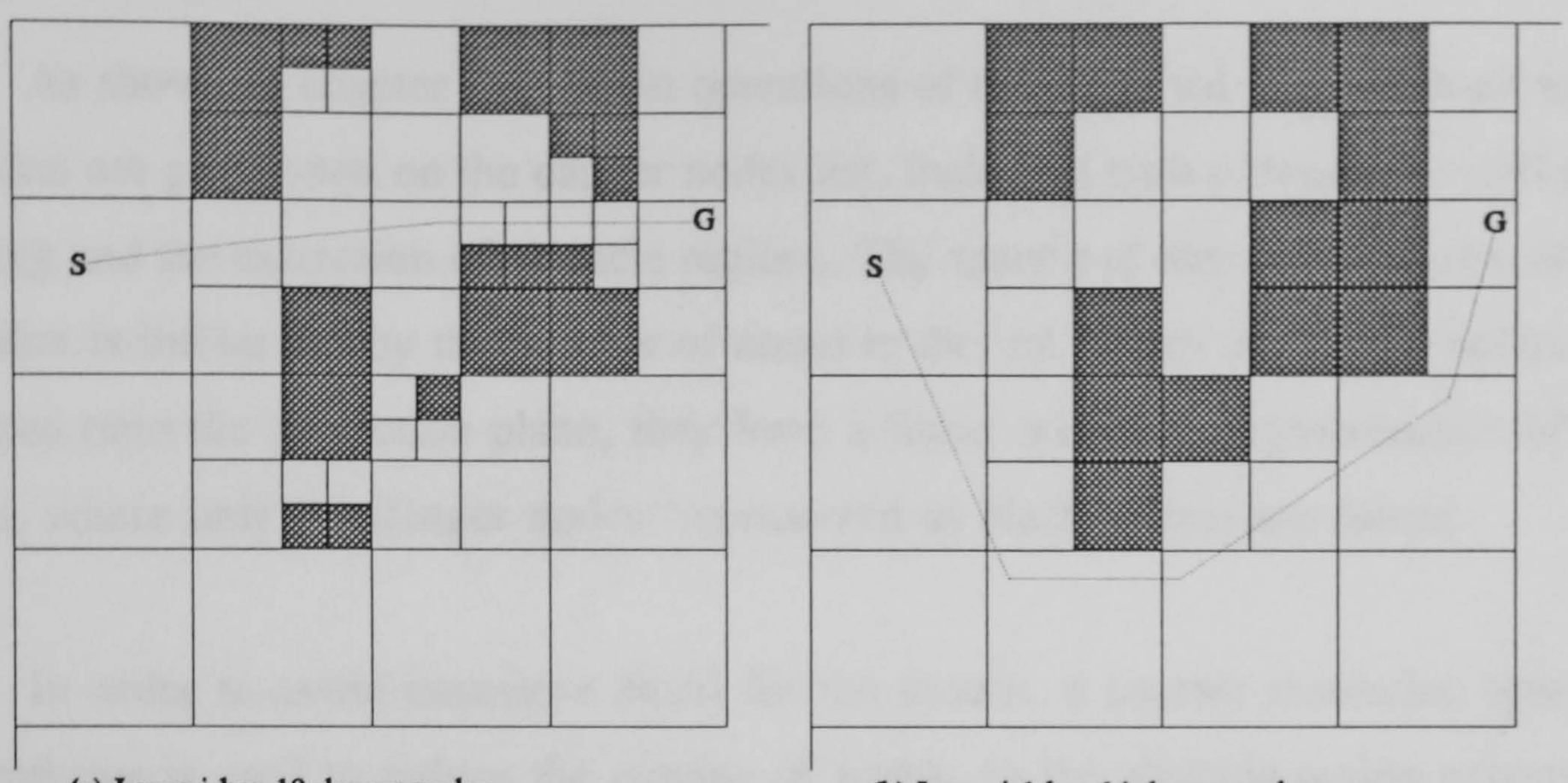
A special feature of the algorithm proposed in chapter four is that the terrain oct-tree only represents the input data. By defining a given flight altitude as a constraint, a preferred path satisfying shortest distance and terrain avoidance constraints is found by using the collision check technique. Before the prototype algorithm in chapter four is applied to a real-time dynamic environment, a pyramid structure of terrain quad-trees is introduced to support the multi-level resolutions of flight path planning.

5.3 Implementation of a Pyramid Structure

5.3.1 Objectives

The navigation space of most cell decomposition methods used in path generation is a grid file of pre-computed cost indices or a grid with overlaid obstacles [Chan85, Lizz85]. In terms of efficiency, the path planning algorithm is initially preset to a coarser resolution level of a terrain oct-tree to prune the searching space. In fact, presetting the size of the cells can result in significant difficulties: a large cell size may prevent a free path from being located while a small cell size clearly increases computation times. For instance, in Figure 5.2, the channel of the path is obscured by the approximation of the large cells in Figure 5.2b; on the other hand, the smaller cell decomposition results in a larger number of grid elements.

Consequently, most methods operate in a hierarchical fashion, by generating an initial coarse decomposition and then locally refining this decomposition until a free path is found [Chan85, Lizz85, Kamb86, Lau87]. The method is conservative and may



(a) Layer i has 19 danger nodes.

(b) Layer i-1 has 14 danger nodes.

Figure 5.2 Path channel obscured by approximation of cells.

fail to generate a free path even if one exists. In the case of failure, it is necessary to refine the decomposition and execute the algorithm again.

In a real-time flight path planning application, a similar strategy is used. It is necessary to avoid the bottleneck of constructing a large visibility graph as this can be the most time consuming process, as observed in the previous chapter. It is also important to prevent the resolution from becoming too coarse to avoid unnecessary blocking of valid paths. In order to achieve a real-time response, a pyramid [Tani75] of quad-trees representing the danger nodes, is used to implement the multi-resolution representation required for the navigation space.

In order to devise the real-time algorithm, it is necessary to examine the multi-resolution of a pyramid structure and the time cost of the algorithm under a static environment. In the next section, the derivation of terrain oct-tree based pyramid structure is discussed.

5.3.2 Multi-resolution Representation of Danger Nodes

As shown in chapter four, most operations of the proposed flight path planning algorithm are performed on the danger nodes list, including such processes as collision checking and the extraction of obstacle regions. The space and time performance of the algorithm is influenced by the number of nodes in the list. When the danger nodes are projected onto the projection plane, they form a linear quad-tree representation of the terrain, where only the danger nodes (represented as black nodes) are stored.

In order to avoid excessive detail for the terrain, a coarser resolution level of the quad-tree is used to reduce the number of nodes. In the obstacle region expansion process discussed in the section 4.4, a coarser level approximation method is applied locally to avoid unnecessary detail during the obstacle expansion process.

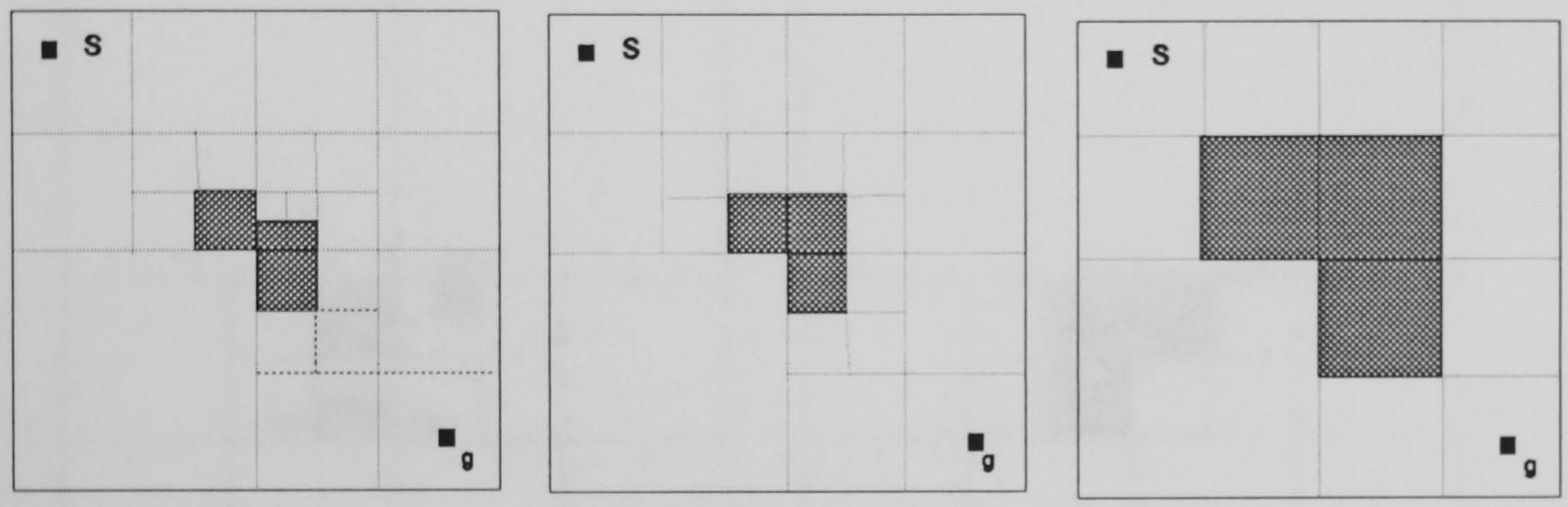
In this section, the terrain oct-tree representation is approximated in a global manner to reduce the number of nodes in a real-time navigation environment. The reduced resolution representation used by the path planning algorithm where a compact representation of the navigation space affords significant gains in space and time efficiency.

A pyramid is defined as a sequence $\{M(i),M(i-1),\ldots,M(0)\}$ of arrays where $M(i)$ represents an original array [Tani80]. The value of a pixel at layer k of the pyramid is a function of the values of the pixels of a m x n 'window area' at layer $k+1$. A pyramid structure of danger nodes consists of a sequence of danger node lists numbered 0 to i, where each layer is actually a linear quad-tree representing the danger area of a given terrain at the corresponding resolution. Layer i is the most detailed (finest resolution) representation; the others are derived directly as shown in Figure 5.3.

The most straightforward method of constructing a pyramid of danger nodes is to apply a reduction rule (e.g. a maximum value function) to each 2 x 2 'window' of the terrain matrix. The maximum elevation values within the window are encoded into

a corresponding locational code of an oct-tree and stored in the north-west quadrant of each window area.

This process proceeds recursively from the level of the original matrix to a single element level. Nodes with scaled elevation values above a threshold at each layer are denoted as danger nodes. In real-time applications, two pyramids are constructed, firstly a pyramid of the terrain oct-tree for data retrieval and secondly, a pyramid of the danger nodes for collision checking.



Layer i, 4 nodes. Layer i-1, 3 nodes. Layer i-2, 3 nodes

Figure 5.3 An example of three-layers pyramid of danger nodes.

As a danger nodes list $M_d(i)$ (obtained from a 2^n x 2^n terrain oct-tree) is arranged in ascending sequence, a pyramid of the danger nodes can be derived directly from the danger nodes list. Each node in the list represents an area of 2^m x 2^m of grid elements, where $0 \leq m \leq n$.

A simple way to obtaining danger nodes at layer *(i-l)* is to increase the size value of the nodes in $M_d(i)$ from m to l ($m < l$). This is achieved by setting the size bits of the last l digits to 1 and all other bits of the last l digits to 0. This process proceeds recursively from level l to a single element level in constructing a pyramid of danger nodes.

For instance, a danger nodes list $M_d(i)$ containing (0218, 0232, 0233, 0301) has four elements generated from a terrain oct-tree. The nodes 0232, 0233, 0301 (m=0, m<l) are approximated to 0238, 0238, 0308 respectively, where node 0218 (m=1) remains unchanged from the list $M_d(i\text{-}1)$. The resultant list is (0218, 0238, 0308). Duplications of 0238 are avoided by checking before a node is appended to the list. The list $M_d(i\text{-}1)$ is a version of list $M_d(i)$ at half the resolution. Figure 5.4 shows above a simple example of this approximation process.

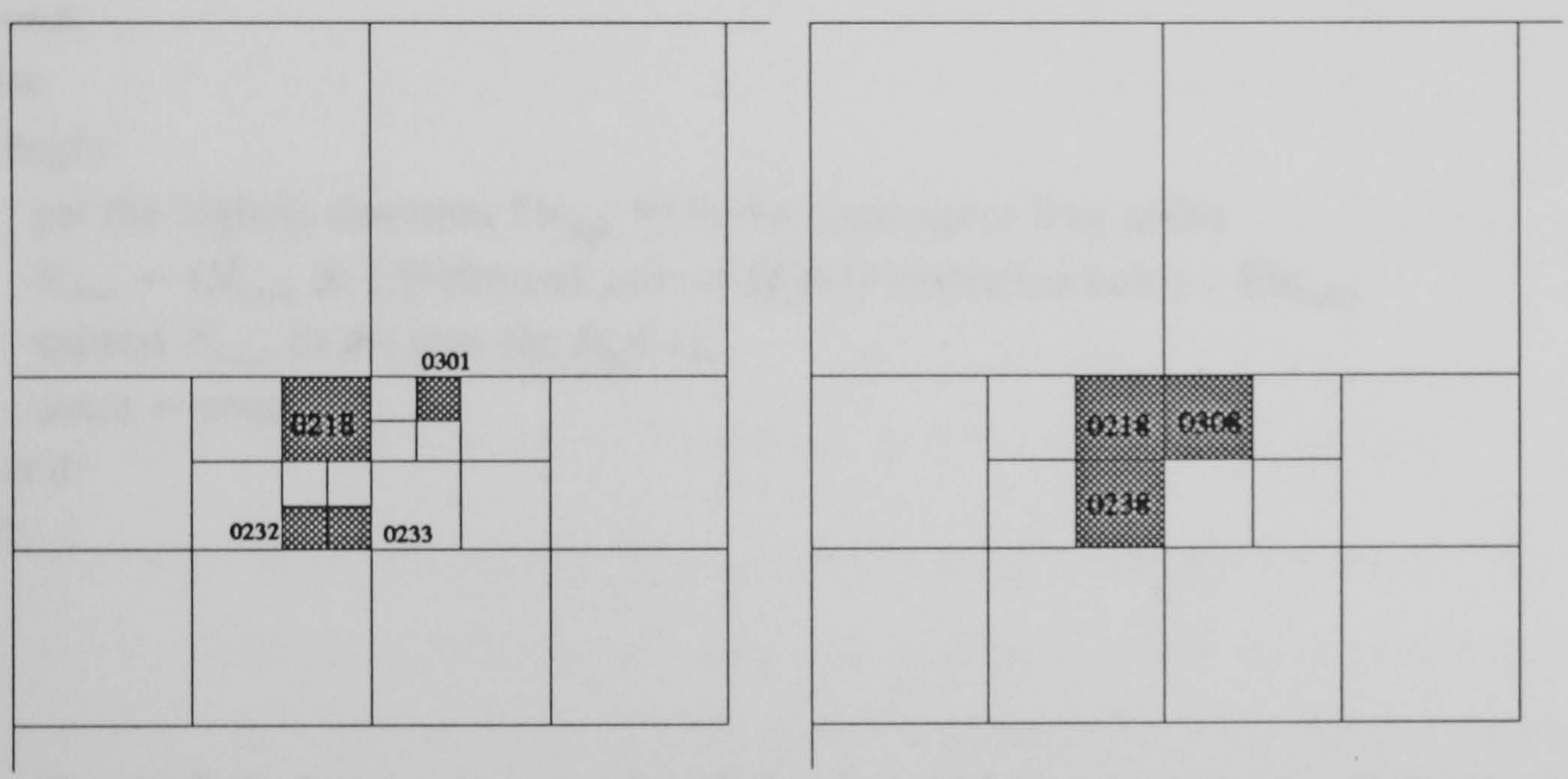


Figure 5.4 Approximation of a danger nodes list.

A pyramid representation of a terrain oct-tree can be derived in a similar way. Each oct-tree in the upper layer of the pyramid has approximately a quarter the number of nodes of its lower layer. The primary advantage of this scheme is that the global structure in a terrain becomes apparent very early in the process, allowing a terrain to be examined at a coarser resolution level. For example, a node at layer (i-l) of a terrain oct-tree is accessed by a window area which covers 2^{m+1} x 2^{m+1} matrix elements and is encoded with the highest elevation value of nodes within the window. The algorithm to derive a nodes list at layer $M_n(i\text{-}1)$ from layer $M_n(i)$ is as follows:

```
procedure LAYER(M_n(i))
begin
  int count ← 0;
  int mask ← bbbbbbbb;
  for every node N_count in the list M_n(i) do
    begin
      check the size of N_count
      if the size is larger than the current resolution unit
        begin
          append N_count to the new list M_n(i-1);
          count ← count+1;
        end;
      else
        begin
          get the highest elevation Ele_high from the consecutive four nodes
          N_count ← (N_count & LShift(mask,size of M_n(i-1) resolution unit)) ¦ Ele_high;
          append N_count to the new list M_n(i-1);
          count ← count+3;
        end;
    end;
end;
```

Figure 5.5 shows an example of the pyramid terrain oct-tree representation. Clearly, the multi-resolution features of pyramid organisation involves an additional storage overhead. The pyramid of a terrain oct-tree requires storage for nodes equal to $N_{node}(1+1/4+1/16+\ldots) = 4/3(N_{node})$, where N_{node} represents the number of nodes at the finest layer of the pyramid.

5.3.3 The Time Performance of Flight Path Planning

To achieve real-time flight path planning, the time performance of the algorithm is measured at different stages of path planning and subsequently used as a reference. The algorithm is coded in the C language and the time performance measurements is based on a 80486 PC system, running at 33 MHz under the MS-DOS operating system.

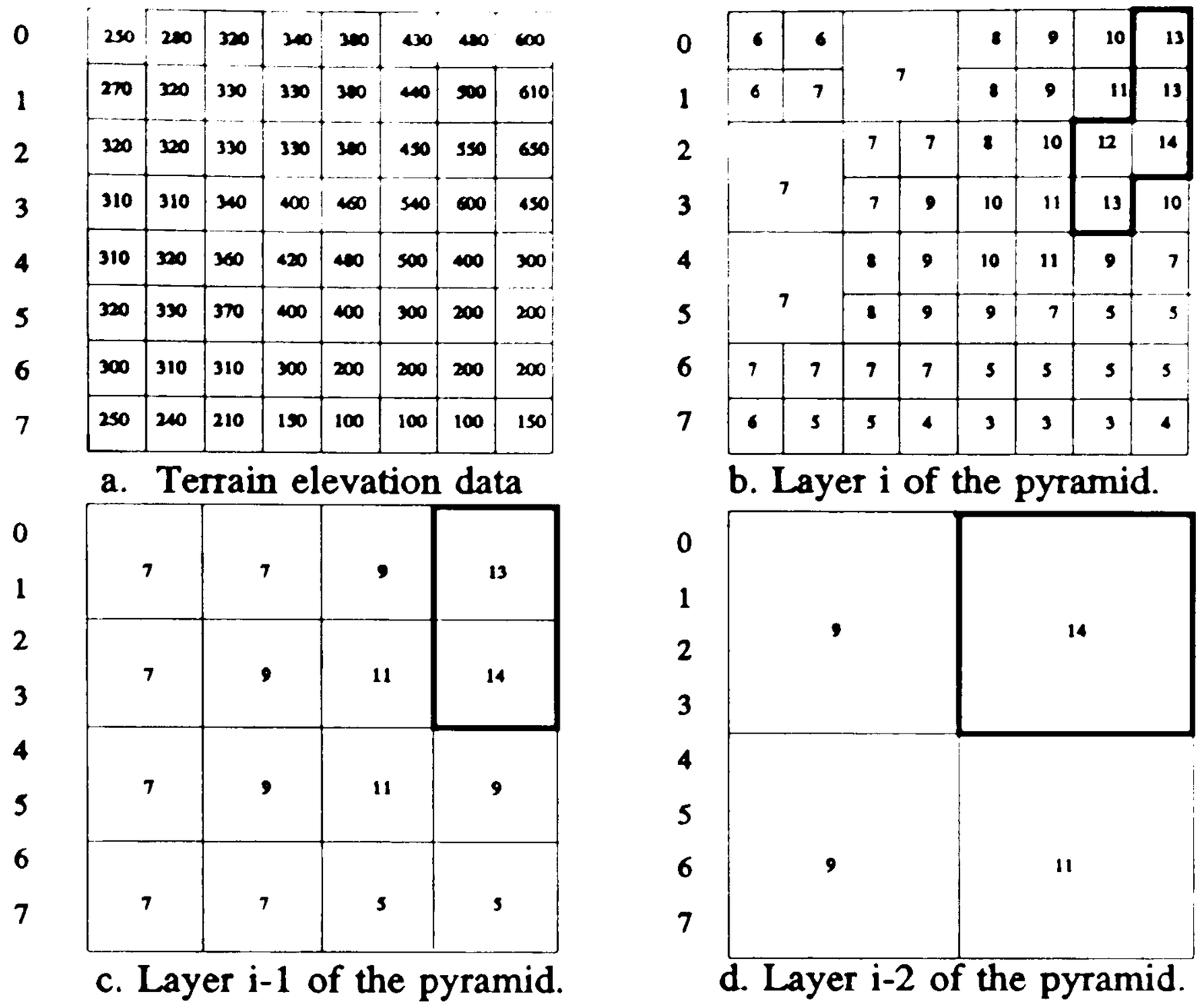


Figure 5.5 An example of pyramid terrain oct-tree derivation.

The algorithm is executed at different resolution layers of terrain oct-tree pyramids. By randomly generating the start and goal points, and taking flight altitude and shortest distance as constraints, the algorithm is exercised to measure the time cost. This time measurement includes:

(1) T_{wp} - the time to obtain the waypoints

(2) T_{vg} - the time to construct the visibility graph

(3) T_{sp} - the time to search for a path of minimum distance

It has been shown in chapter 4 that the maximum times for the obstacle expansion function are $O(4^{d+1}*(p+n))$ and is proportional to the perimeter of the obstacle regions. Once the waypoints are located in navigation space, the time performance of the subsequent collision check based visibility graph construction is proportional to the path segments, where each collision check depends on the distance of a path segment. In the graph search stage, a depth-first or breadth-first search method (described in section 4.6) takes $O(r)$ time whereas an A^* search takes

$O(r+nlogn)$, where r is the number of edges and n is the number of nodes in the visibility graph.

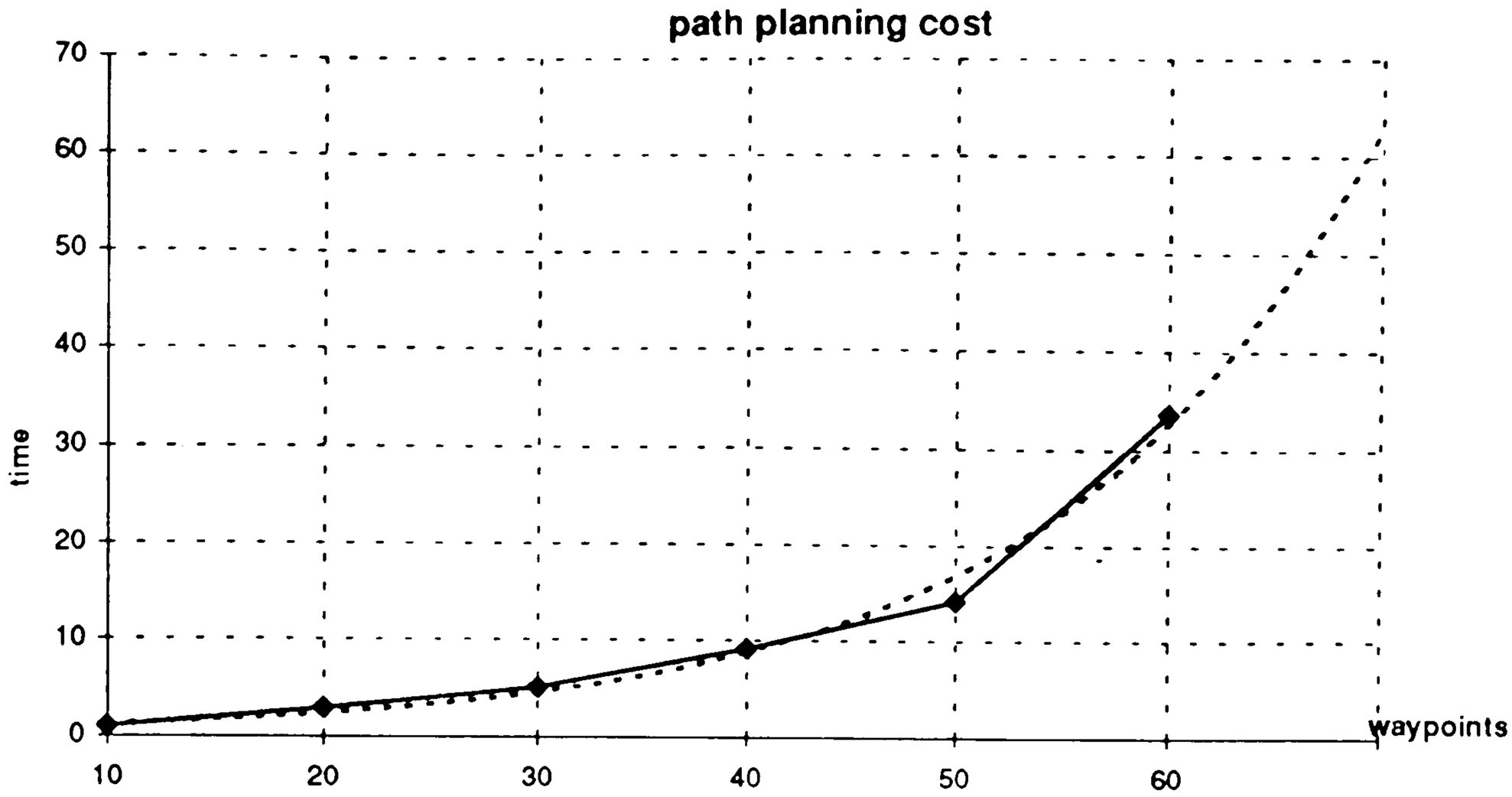
As the obstacle regions are defined by the minimum flight altitude and the track of direct flight of the aircraft, there is no regularity of the obstacle profile of different terrain areas and it is therefore difficult to derive a general representation of time performance which can cover a range of terrain cases. In order to predict the trend of the time performance of the flight path planning algorithm, the experimental results are evaluated according to the range of the number of waypoints and the corresponding mean time cost of each path planning stage.

Table 5.1 presents a summary of observations based on performing the path planning algorithm 1000 times on each of the 10 terrain oct-trees which are encoded from two DTM files. The start and goal points are randomly generated within the terrain oct-trees navigation space. The relationship between the number of waypoints and the time cost of each path planning stage is used as the index of performance.

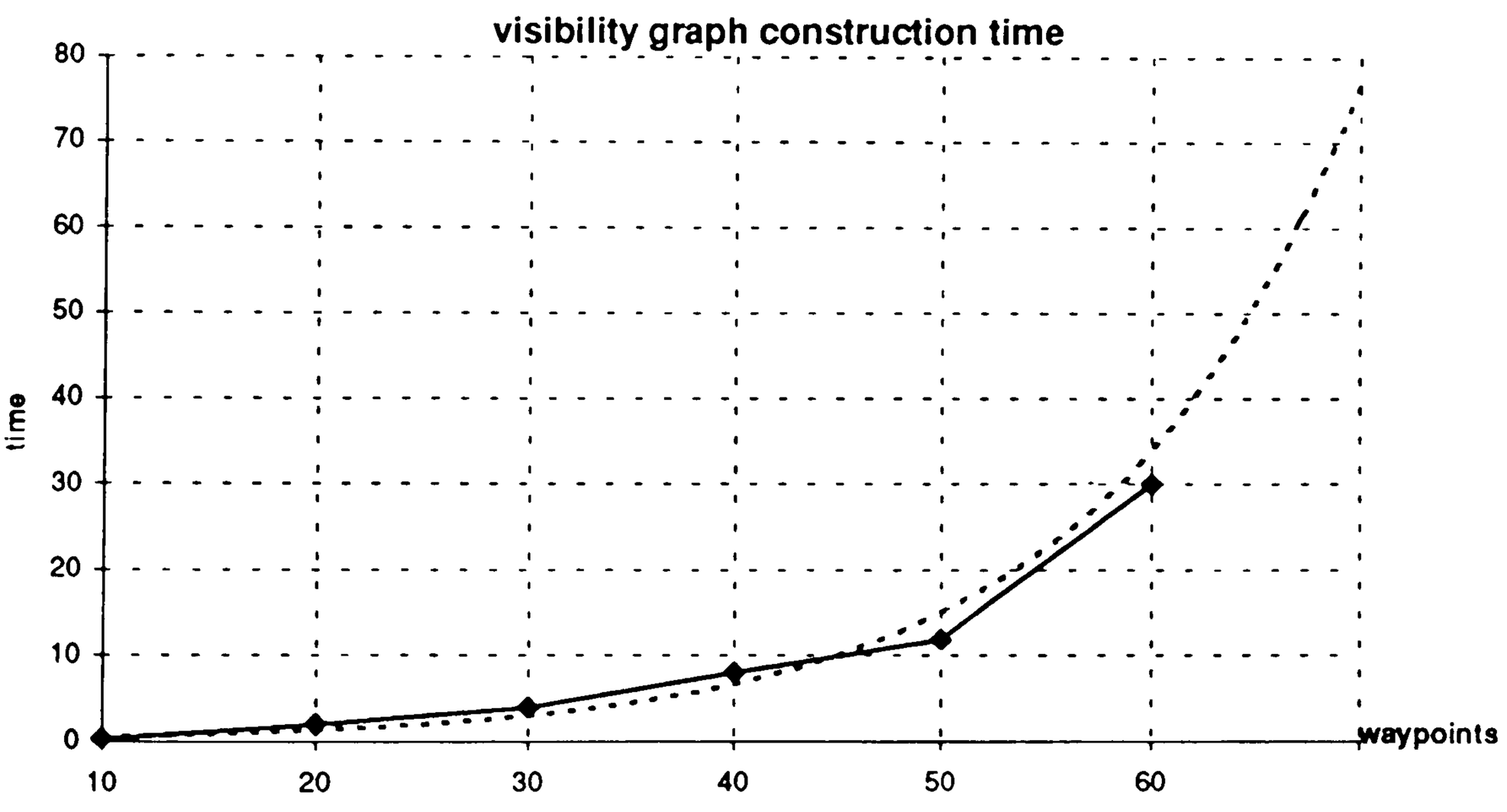
No of waypoints	T_{WP} (sec)	T_{VG} (sec)	T_{SP} (sec)	T_{total} (sec)
< 10	0.055	0.330	0.604	0.989
11 - 20	0.165	1.978	0.714	2.857
21 - 30	0.440	3.901	0.714	5.055
31 - 40	0.440	7.967	0.714	9.121
41 - 50	0.549	11.758	1.346	14.120
50 - 61	1.813	30.055	1.813	33.671

TABLE 5.1. The summary of the mean time cost for each stage of the flight path planning algorithm based on 10,000 tests of 10 terrain oct-trees constructed from DTM files.

Figure 5.6 shows how the time cost of different stages of the path planning increases with the number of waypoints. However, Table 5.1 shows that the cost of obtaining the waypoints (T_{wp}) and searching for a path in a visibility graph (T_{sp})



(a) Total cost - Waypoints diagram



(b) Visibility graph construction time - Waypoints diagram

Figure 5.6 The time performance of the flight path planning algorithm and the prediction of the cost with respect to the number of waypoints in the navigation space.

constitutes a small amount of time in comparison with the cost of the total path planning process.

The experimental results in Table 5.1 show that the time T_{vg} for the visibility graph construction determines whether the predicted delay is acceptable for real-time applications. The cost T_{vg} is $O(W^2*(N_{lp}*logN_{dn}))$ (refer to section 4.5.2) which corresponds to the number of times the binary search is executed in collision checking during the construction of the visibility graph. In the T_{vg} cost expression, W is the number of waypoints, N_{lp} is the number of points in a path segment and N_{dn} is the number of danger nodes, as described in section 4.5.2.

For example, in Figure 5.7, a path segment between points S and G with 14 nodes at layer *i* may represented by 8 nodes at layer *i-1*. The number of danger nodes at layer *i* may only be one quarter of the number of danger nodes at layer *i-1*. It is clear that the time performance of the flight path planning algorithm can be improved by reducing the resolution layer of navigation space.

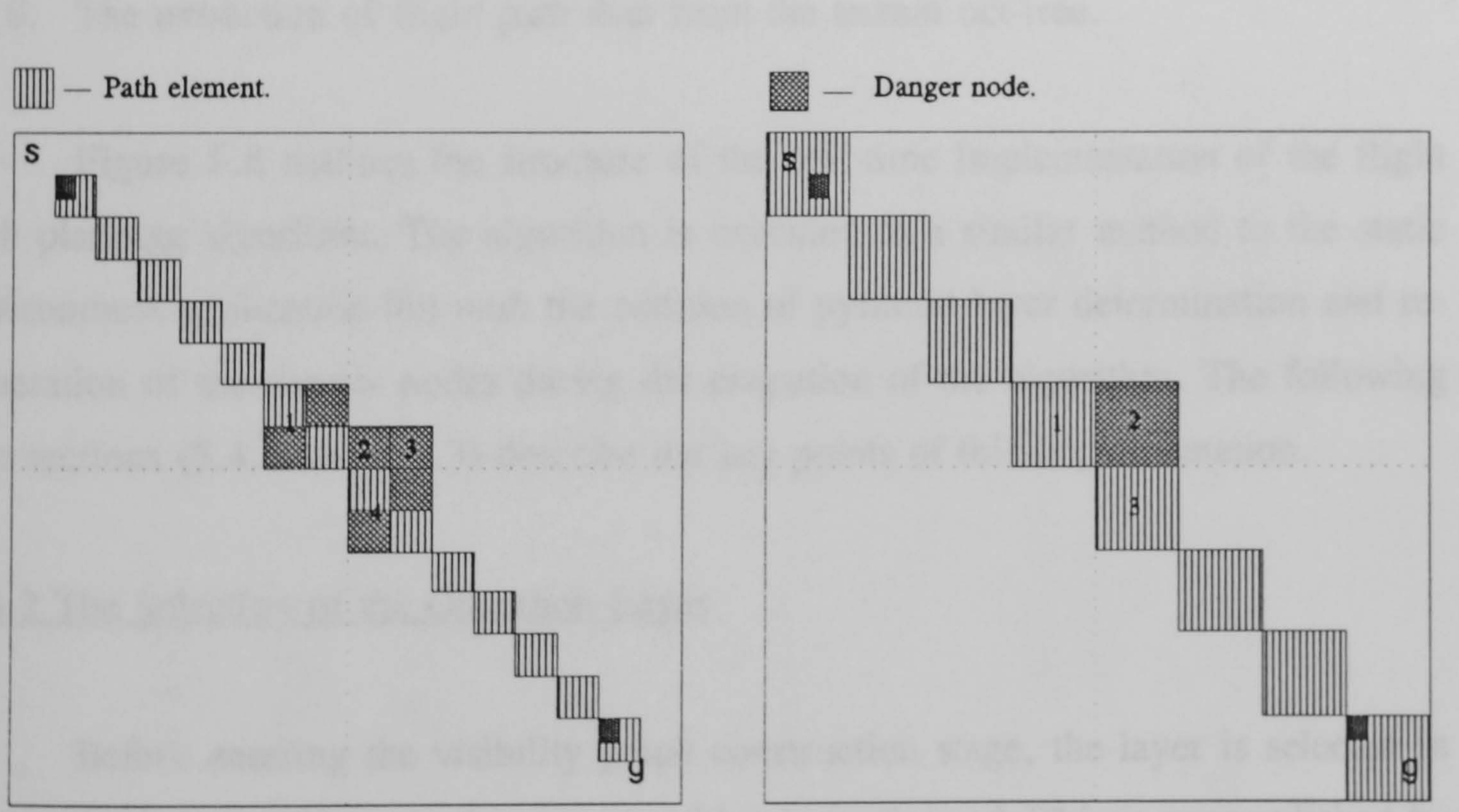


Figure 5.7 An example of collision check at different resolution layer.

5.4 Real-time Dynamic Flight Path Planning

5.4.1 Algorithm Structure

The real-time flight path planning algorithm generates reference trajectories subject to constraints of flight altitude, path distance and time to acquire the path. The algorithm includes following stages:

1. The physical location of the new start point.
2. The extraction of the new list of danger nodes if the flight altitude is changed.
3. The computation of waypoints according to the new destination.
4. Determination of the resolution layer according to the number of waypoints.
5. The construction of the visibility graph.
6. The path searching process.
7. The reduction of the resolution layer if a path is not found.
8. The extraction of flight path data from the terrain oct-tree.

Figure 5.8 outlines the structure of the real-time implementation of the flight path planning algorithm. The algorithm is executed in a similar method to the static environment application but with the addition of pyramid layer determination and re-generation of the danger nodes during the execution of the algorithm. The following two sections (5.4.2 and 5.4.3) describe the key points of this implementation.

5.4.2 The Selection of the Operation Layer

Before entering the visibility graph construction stage, the layer is selected in the pyramid where the path planning algorithm is performed. This is accomplished by obtaining the waypoints in a predefined layer and then determining whether the process should be continued or restarted at a coarser layer relative to the current layer.

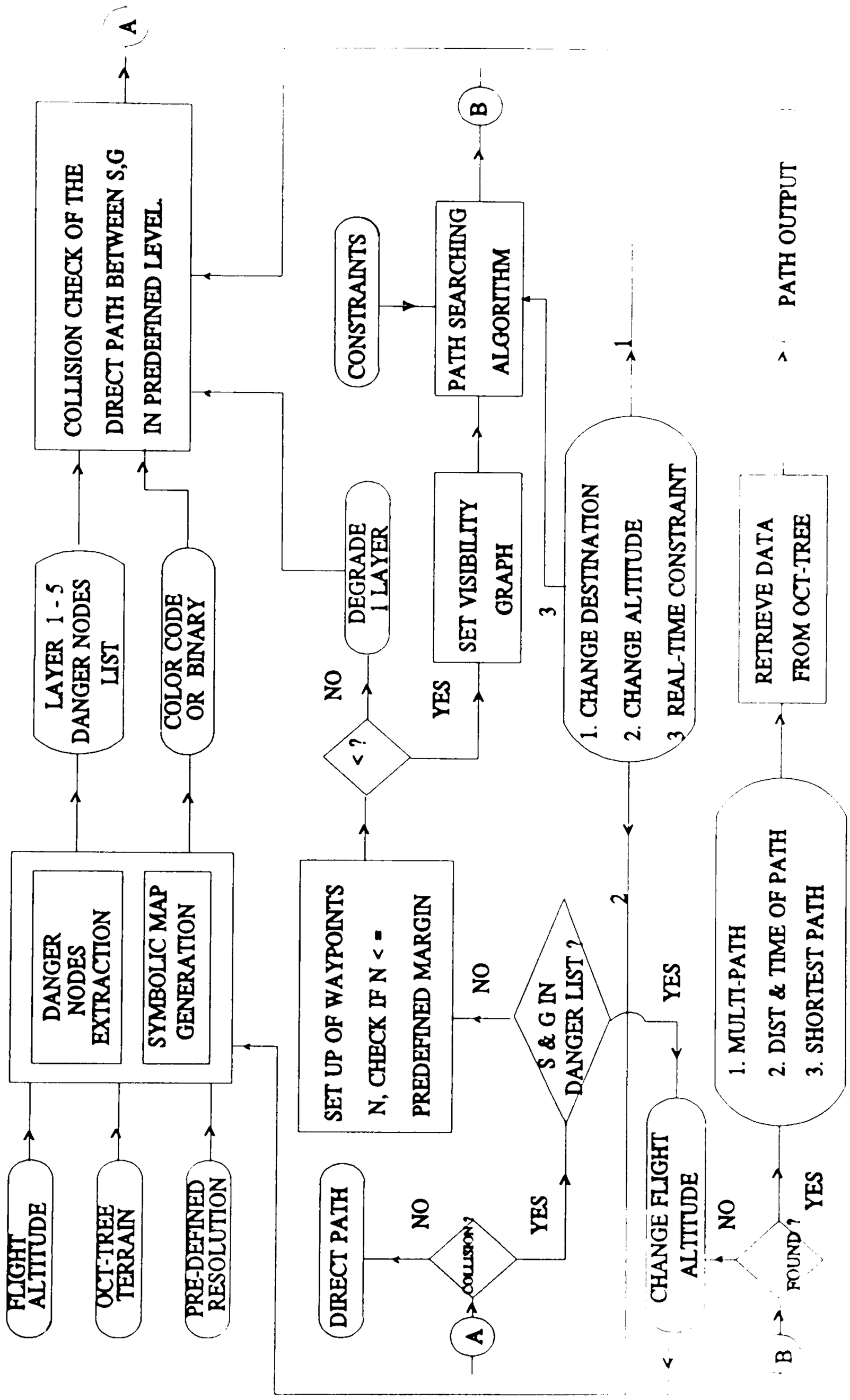


Figure 5.8 The structure of the real time implementation of the flight path planning algorithm.

If flight altitude is one of the flight path constraints, the number of corresponding danger nodes reduces as the flight altitude increases. If there is no path from the start point to the goal point at the current altitude, the algorithm will try to find a path at a higher flight altitude and will eventually reach an altitude which contains no obstacles.

There are two cases in which the visibility graphs have to be reconstructed:

1. The danger nodes vary in relation to the minimum flight altitude. Whenever the minimum flight altitude is changed, the replanning process starts from the construction of a pyramid of the danger nodes.
2. If the goal point is changed but the flight altitude remains unchanged, the original pyramid can be re-used. A new set of obstacle nodes is extracted corresponding to the new goal point.

Initially, the path planning process starts at layer (*i-2*) of the pyramid of terrain oct-trees. The resolution unit at layer *i* of the oct-tree is 100 metres, whereas layer *(i-2)* is 400 metres. From observation, layer (*i-2*) is the layer where most paths are found under the constraints of time and shortest distance. The total time cost of path planning is generally less than three seconds.

5.4.3 The Time Constraint

Before the flight path planning algorithm is operated in a real-time environment, and since an acceptable figure for a real-time application depends on the computer systems and the aircraft speed, the requirement is simplified to five seconds in this thesis. As mentioned in section 5.2, assuming the planning process takes 5 seconds to find a path, an aircraft must fly along current path for 5 seconds before a new path can be obtained. The new start point is estimated at a distance *(5*aircraft speed)* ahead of the point where the request for a new path was issued.

In order to ensure that a path is found before the aircraft reaches the new start

point, the visibility graph is only constructed at layer (*i-2*) or above. Table two shows that the total time cost at layer (*i-2*) is less than three seconds with a maximum of 20 waypoints, and is always less than one second at layer (*i-3*).

The above observations imply that a realistic constraint of executing the flight path planning algorithm is five seconds. This period of five seconds allows the full scale path planning process to be performed twice at most for layers (*i-2*) and (*i-3*) where the resolution is 400 and 800 metres respectively.

Although, these results do not represent the general case, the number of waypoints is only used as a guideline for the real-time implementation. The time performance depends on the computing system. Furthermore, the number of nodes generated by the terrain oct-tree construction algorithm (discussed in chapter three) depends on the terrain source data and the scaling factor and the number of danger nodes and obstacle nodes depend on the minimum flight altitude. Nevertheless, the implementation of the pyramid structure provides an efficient method of determining an appropriate navigation space.

5.4.4 Algorithm Description

So far, the algorithm structure of the real-time implementation of flight path planning, the selection of the operation layer and the time constraints have been discussed. The sequence of processes can be described as follows:

1. Each node of a terrain oct-tree is checked according to a given minimum flight altitude to extract a list of danger nodes. A pyramid of danger nodes is constructed from this list of danger nodes.

2. Layer three is predefined as the operation layer for path planning. By using the Bresenham's line generating algorithm, a list of 'seed' nodes is derived along the line between the start and goal points. The collision checking is applied to

the 'seeds' to locate the vertices of obstacles and then to locate a set of waypoints. If either the start or the goal point falls in the danger area, a new pyramid is generated according to the scaled elevation of the start or goal point. If no collision happens, the planning process is terminated with a direct flight path.

3. If the number of waypoints is greater than 20 in the current layer, then the planning process degrades to layer four and the waypoints are re-generated. The degrading of layers will continue until a layer is reached where the number of waypoints is less than or equal to 20.

4. Path searching is then performed on the visibility graph constructed by the waypoints. If a path is not found in the visibility graph, then the planning process is repeated for a new pyramid of danger nodes by changing the minimum flight altitude. The new pyramid can be obtained directly from the old one. If the full scale planning process is performed twice and a path is still not found, a direct path between the start and goal points is suggested with a minimum flight altitude.

As discussed in section 5.4.3, the real-time application depends on the performance of the computing system and the aircraft speed and the real-time constraint. The time allowed for performing the planning and the predefined layer of process are variable. The multi-resolution feature of a terrain oct-tree pyramid affords the flexibility.

5.5 Path Searching

In the visibility graph, edges can be labelled by costs where the cost of a path is defined as the sum of the costs attached to its edges. Such costs can be defined as the distance of each path segment [Lizz85] or by a penalized cost function attached to a path segment which relates to a known threat [Chan85]. In this case, a minimum-cost

path is generated between the start point and the goal point.

As shown by the heuristic search method in section 4.7, the cost model is simplified to the shortest flight distance. The least-cost distance and hill-climbing methods are used to find an optimal solution in that section. A variant of the A* algorithm is also implemented with the evaluation function *f* of a node *N* defined as $f(N) = g(N) + h(N)$ to find a shortest flight path, where $g(N)$ represents the distance of the path segments from the start point to *N*, $h(N)$ represents the heuristic estimate of the distance of the remaining path from *N* to the goal point which is calculated as the Euclidean distance between *N* and the goal node. The A* algorithm implementation is given in appendix one.

All of the search techniques described so far are concerned with finding an individual path. In aircraft navigation applications, it is useful to find several possible solutions between the start and goal point. Multiple solutions can present various options to help a pilot to select the best solution for his current situation, or to match different constraints or requirements during the flight.

The method used is to remove from the visibility graph the solutions that are already found and then to attempt to find another solution. This method actually prunes limbs from the graph. Because any connection that is part of a solution will have its backtracking field **skip** marked, it can no longer be found by the function FIND( ) (discussed in section 4.6). Hence, all connections in a solution are effectively removed. It is necessary to clear the backtrack stack for the next run of the path planning process.

An alternative way of obtaining an optimal path is to find a best path among those solutions by employing multiple solution generation techniques. For example, if a shortest distance path is required, it can be extracted by combining the path removal methods of generating multiple solutions with the least-cost search (refer to section 4.7) in order to minimize the distance.

The key to finding the shortest path is to retain a solution with a distance that is less than the previous one. When the program cannot generate any further solutions, the optimal solution remains. To accomplish this, a backtrack stack is used which contains the path segments obtained from the path searching routine together with a solution stack which holds the current solution.

By using the graph search function DEPTH_SEARCH or BREADTH_SEARCH discussed in section 4.6, function SHORTEST_PATH keeps the shortest path among the multiple solutions in the resolution stack, where function PATH_OUTPUT gives the shortest path as well as the multiple paths. The algorithms are shown as follows:

```
procedure SHORTEST_PATH( )
begin
  value integer dist, t;
  value integer tos, stos; counter of backtrack and solution stack;
  value integer old_dist ← any value large than the boundary of gaming area;
     begin
       if backtrack stack is empty, return 0;
     begin
       t ← 0;
       dist ← 0;
       while (t < tos), backtrack stack is not empty do
         begin  read out the path segment data;
           dist ← accumulate the distance of path segments from the backtrack stack;
           t ← t+1;
         end;
     end;
if new path distance is shorter than the old path distance
    begin make new solution
       t ← 0;
       old_ dist ← dist;
       stos ← 0, clear old path data from solution stack;
       while (t < tos), backtrack stack is not empty do
         begin
          copy the path segment data from backtrack stack to solution stack;
          t ← t+1;
         end;
    end;
```

```
      return dist;
    end;
end;

procedure PATH_OUTPUT(start, goal)
begin
  value integer d, t;
  value integer tos, stos; counter of backtrack and solution stack;
  SETUP(W_points); set up list of path segment;
if multi-path wanted
  begin
    while (d != 0) still finding solutions do
      begin
        DEPTH_SEARCH(from, to);
        d ← SHORTEST_PATH( );
        while (t < tos) do
          begin
            output the solution from backtrack stack;
            t ← t+1;
          end;
        tos ← 0;
        t ← 0;
      end;
  end;
if shortest path wanted
  begin
    while (d != 0) still finding solutions do
      begin
        DEPTH_SEARCH(start, goal)
        d ← SHORTEST_PATH( );
        tos ← 0;
        t ← 0;
      end;
    while (t < stos) do
      begin
        output the shortest path from solution stack;
        t ← t+1;
      end;
  end;
end;
```

5.6 Summary of The Real-time Implementation

Real-time dynamic flight path planning is achieved by using an hierarchical multi-resolution terrain model to provide the necessary computational efficiency. A resolution of the terrain representation can be degraded or upgraded as required. A pyramid of the danger nodes provides a compact representation of navigation space where the path planning process is able to switch between the different resolution layers. A preferred resolution layer is then selected as the domain of the path planning process.

A key point of the real-time implementation is the constraint on the time to obtain a new path. Five seconds is allowed based on performance measurements of the algorithm in a static environment. The performance also depends on the hardware system, the software implementation and the speed of an aircraft. The time constraint defined in this chapter is given as a demonstration of real-time applications.

The algorithm generates a preferable flight path, the time and distance to fly and the terrain elevation along the flight path. Because the obstacles nodes are located along a flight path, the flight path planning algorithm provides obstacle collision avoidance by restricting the aircraft to a known safe flight altitude.

Although the path planning process is performed at a coarser resolution layer of a pyramid, because a node is encoded with its north-west corner coordinates and the highest elevation inside a 2^m x 2^m window area, the planar position is fixed at the finest resolution level and the vertical resolution is given by the scaling factor which is adopted in the encoding process of a terrain oct-tree. The results imply that the position accuracy of a terrain oct-tree based flight path is the same as its DTED source file. The nodes along a path form a 'corridor' above a given minimum flight altitude which provide a flexible flight path within the coverage area.

In this chapter, the strategy of applying the proposed flight path planning

algorithm to a real-time dynamic environment has been discussed. The off-line results which including the time to generate the waypoints, the time to construct a visibility graph and the time to find a path at different resolutions of the terrain oct-tree are used as references for 'tuning' the real-time dynamic flight path planning algorithm for a given terrain.

In the next chapter, the experimental results of terrain oct-tree encoding and decoding, the static mode flight path planning algorithm and real-time dynamic implements are described. The results demonstrate that the terrain oct-tree representations and applications can be integrated in a TRN system.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 System Description

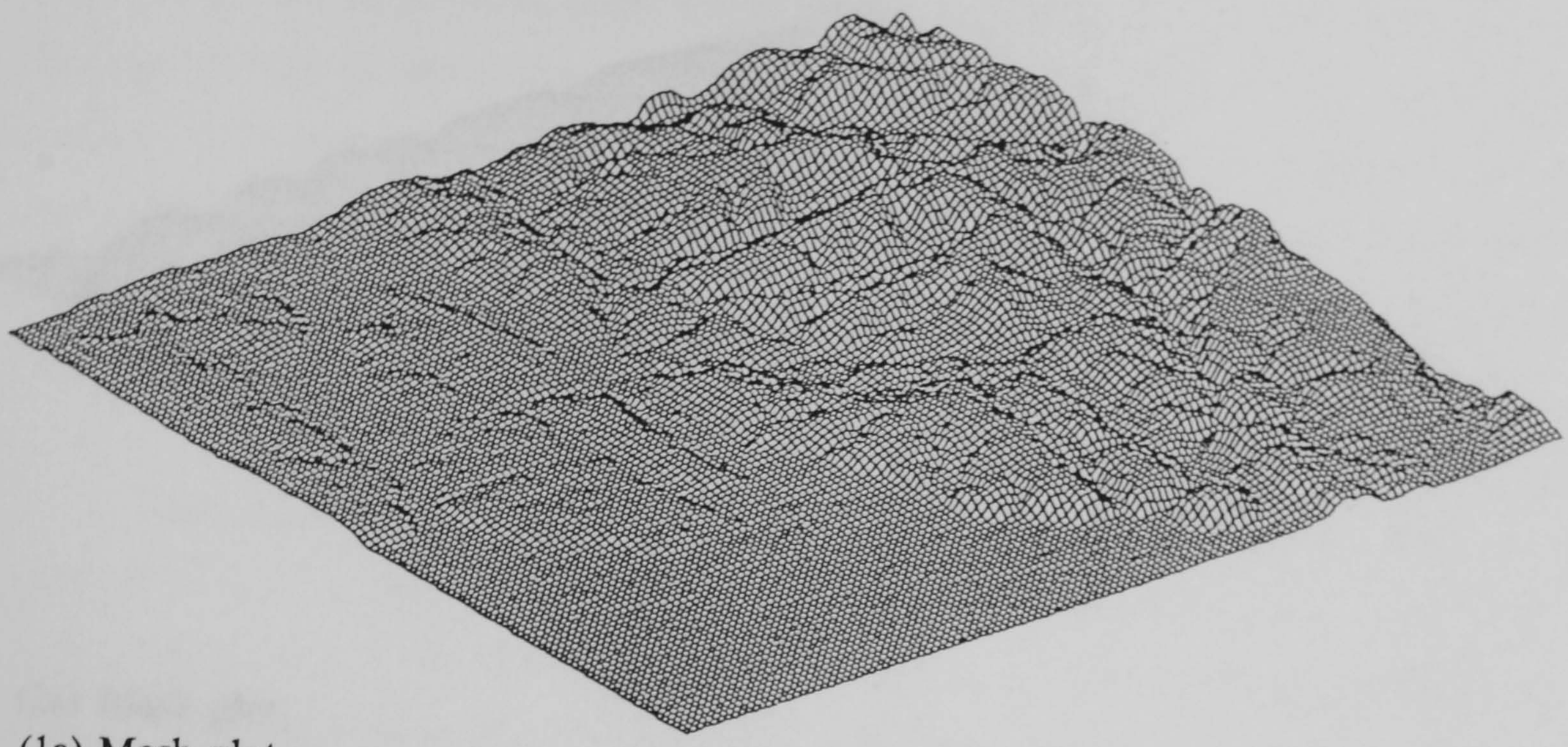
The terrain oct-tree data structure and flight path planning scheme described in the previous three chapters enable the real-time generation of flight paths for airborne navigation. The flight path planning algorithms consist of two parts:

- In the design of terrain data structures, the algorithms include the encoding and decoding of the locational code, the construction of the terrain oct-tree, the construction of the terrain pyramid and the terrain data retrieval.

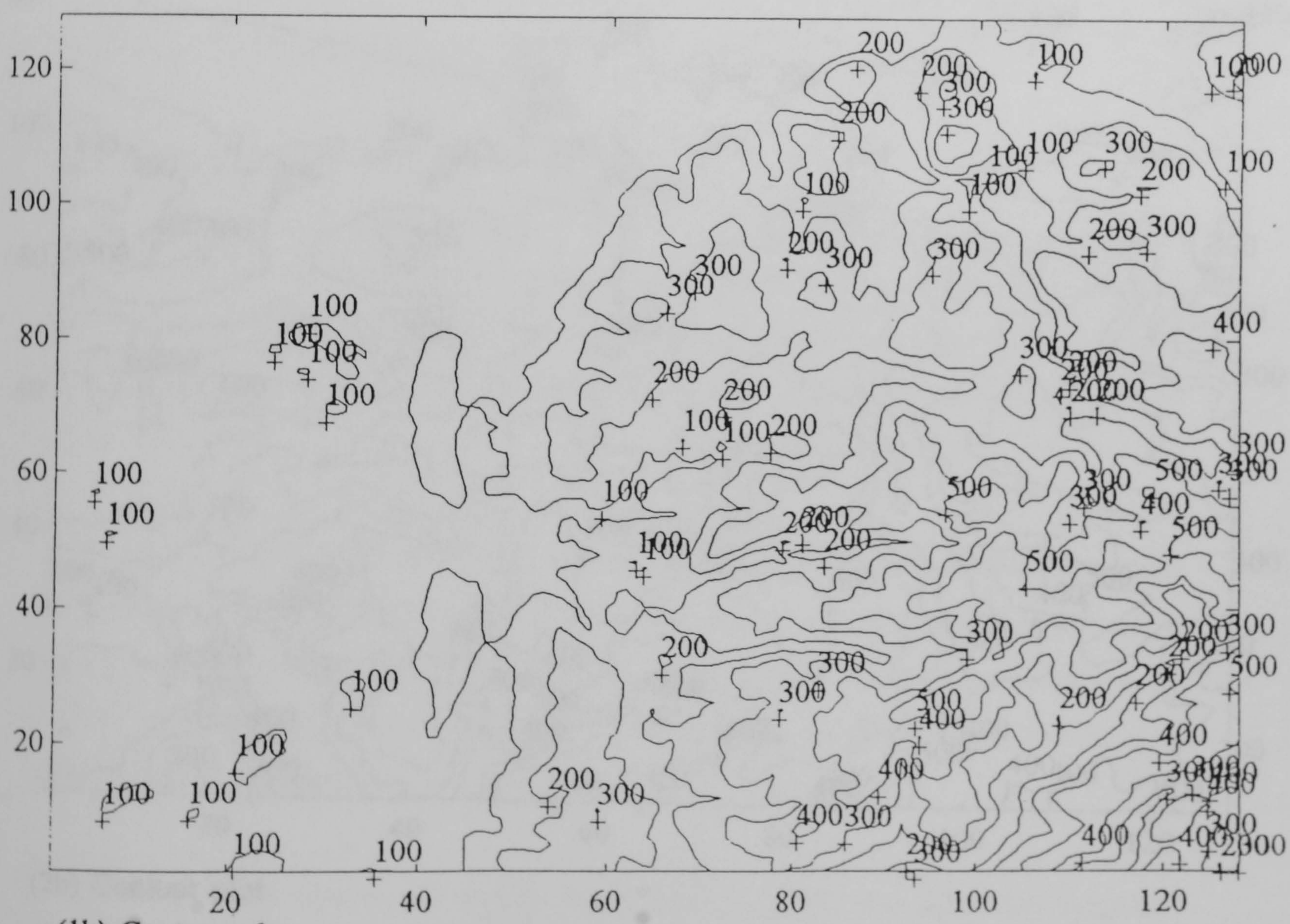
- In flight path planning, the algorithms include obstacle region expansion, collision checking, the construction of the visibility graph and path searching.

In order to extend the application of the path planning algorithm from a static environment to a real-time dynamic implementation of flight path planning, the algorithm is applied to a set of terrain oct-trees to observe the time performance of the algorithm. One of the most important aspects of these algorithms is the encoding of a terrain oct-tree from digital terrain elevation data.

The digital terrain elevation data used in this study is the OSGB 1:50000 Digital Terrain Model Data (DTM) obtained from the Ordnance Survey. The DTM file consists of height values at intersections of a 50 metre horizontal grid. The DTM file is defined as a matrix of size 601 x 601 covering 30 square Km. One set of DTM files has been provided for an area of the Peak District, the other set of DTM is for an area around Port Talbot in Wales. Figures 6.1 and 6.2 show the contour and mesh plot of these files.

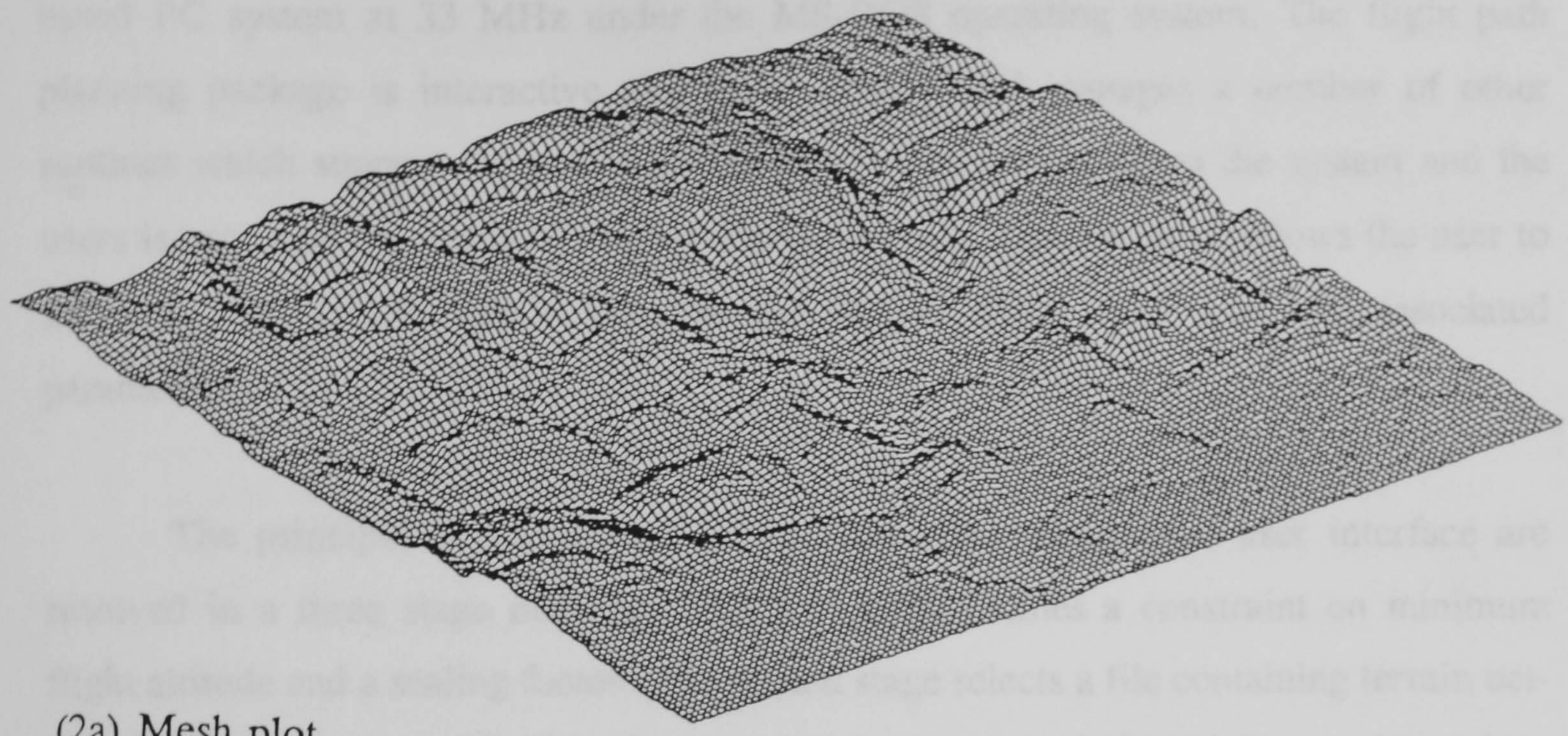


(1a) Mesh plot.

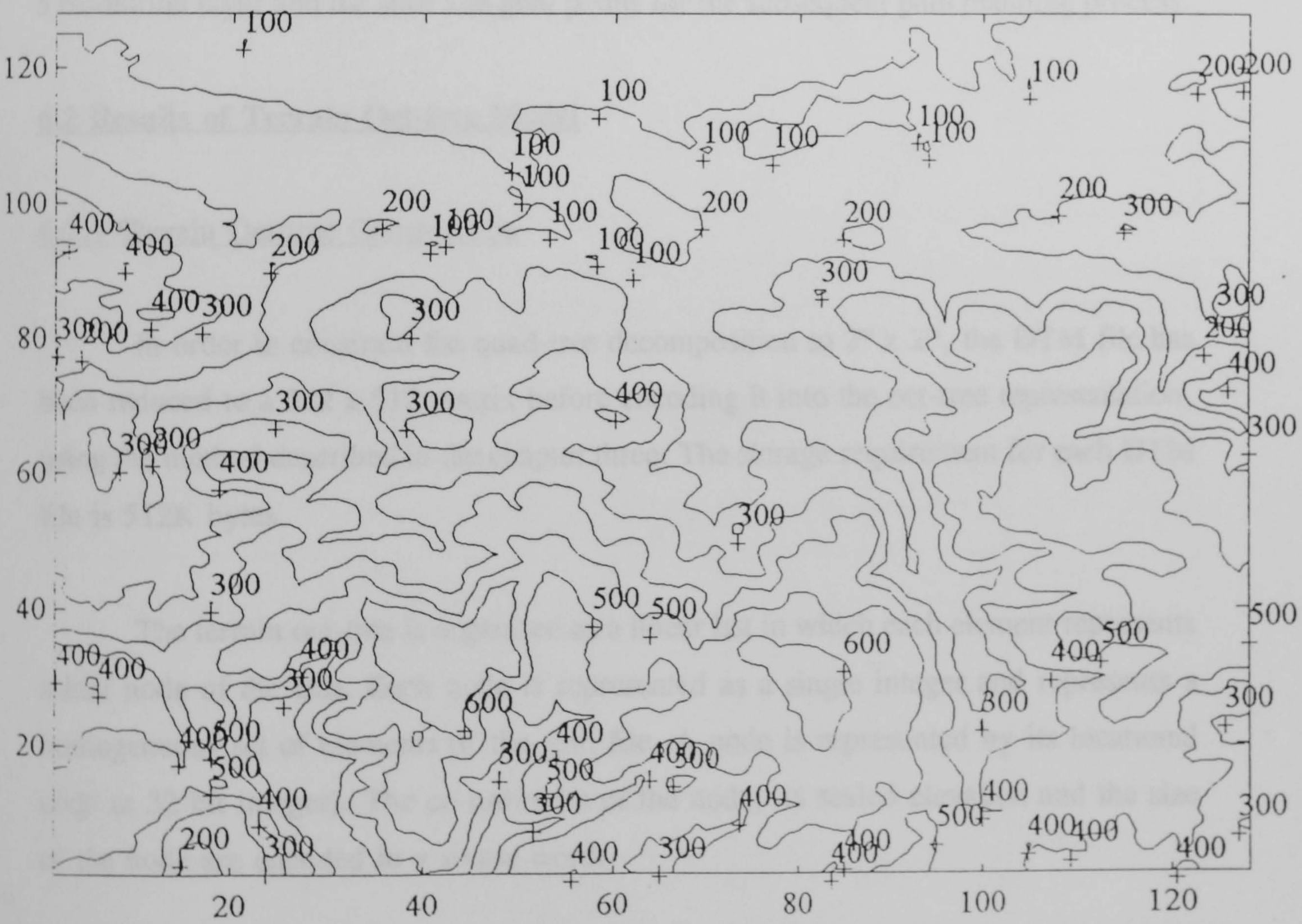


(1b) Contour plot.

Figure 1 1:50000 DTM data , coverage of 25.6km x 25.6km around Port Talbot area.



(2a) Mesh plot.



(2b) Contour plot.

Figure 2 1:50000 DTM data , coverage of 25.6km x 25.6km around Peak District area.

The software is written in the C programming language, running on an 80486 based PC system at 33 MHz under the MS-DOS operating system. The flight path planning package is interactive and menu driven, and manages a number of other routines which support the path planning tasks. Dialogue between the system and the users is provided by means of keyboard and a mouse. This dialogue allows the user to select an appropriate terrain oct-tree and defines flight conditions and associated parameters.

The principal characteristics of the interactive menu-drive user interface are resolved in a three stage dialogue. The first stage defines a constraint on minimum flight altitude and a scaling factor. The second stage selects a file containing terrain oct-tree data. After a pyramid of the danger nodes is constructed, the third stage determines a resolution layer and the start and goal points for the subsequent path planning process.

6.2 Results of Terrain Oct-tree Model

6.2.1 Terrain Oct-tree Construction

In order to constrain the quad-tree decomposition to 2^m x 2^m, the DTM file has been reduced to a 512 x 512 matrix before encoding it into the oct-tree representation, using the method described in the chapter three. The storage requirement for each DTM file is 512K bytes.

The terrain oct-tree is organised as a linear list in which each element represents a leaf node of the tree. Each node is represented as a single integer and represents a homogeneous set of elements of the grid file. A node is represented by its locational code (a 32 bit integer). The co-ordinates of the node, its scaled elevation and the size of the node are encoded in a single word.

Table 6.1 compares the number of nodes in a set of 64 x 64 (4096) synthetic terrain encoded grid files for both the standard linear oct-tree encoding method

[Garg82c] and the new terrain oct-tree encoding method developed in chapter three. It is shown in Table 6.1 that there is no significant gain for the linear oct-tree encoding method in terms of the reduction in the number of nodes compared with 4096 grid file elements. However, the number of nodes in the terrain oct-tree method, which uses a quadrant merging criteria in projection space, is significantly decreased.

Scaling Factor	Number of Nodes		Improvement (%) ([A]-[B]) / [A]
	Linear Oct-tree [A]	Terrain Oct-tree [B]	
50	3688	3463	6.1%
75	3433	3105	9.6%
100	3442	2584	24.9%
125	3160	2473	21.7%
150	3091	2182	29.4%
175	3022	1888	37.5%
200	3040	1585	47.8%

TABLE 6.1. The experimental results of the linear and terrain oct-tree encoding methods using different scaling factors. Sample terrain file elevations range from 0 to 750 metres.

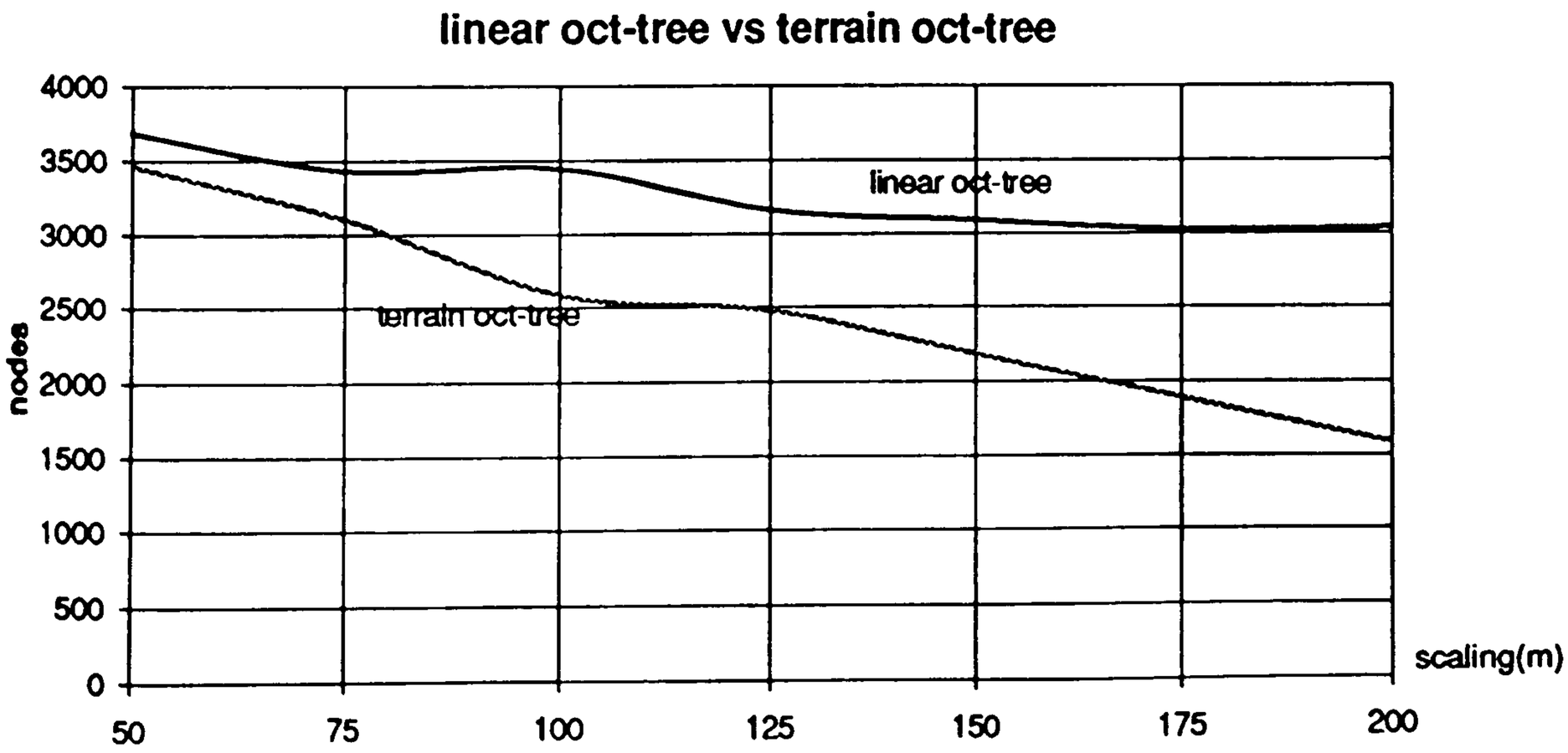
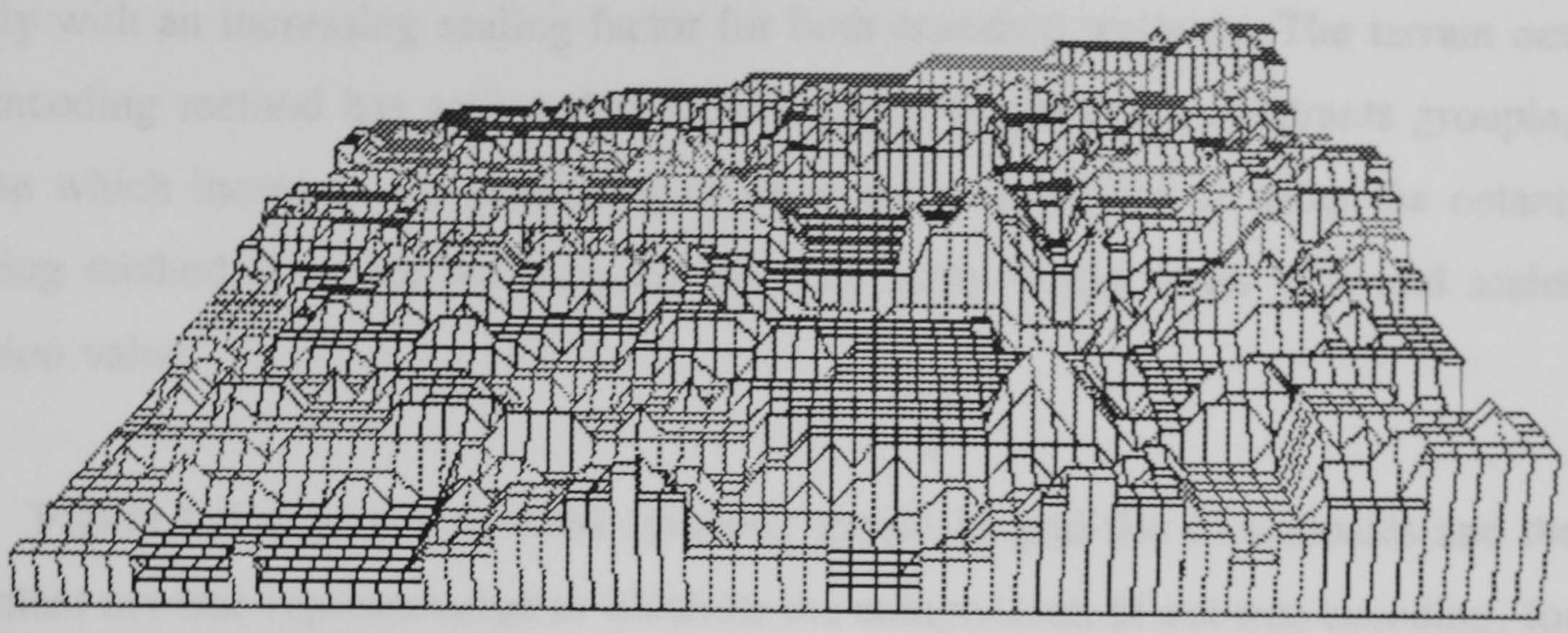
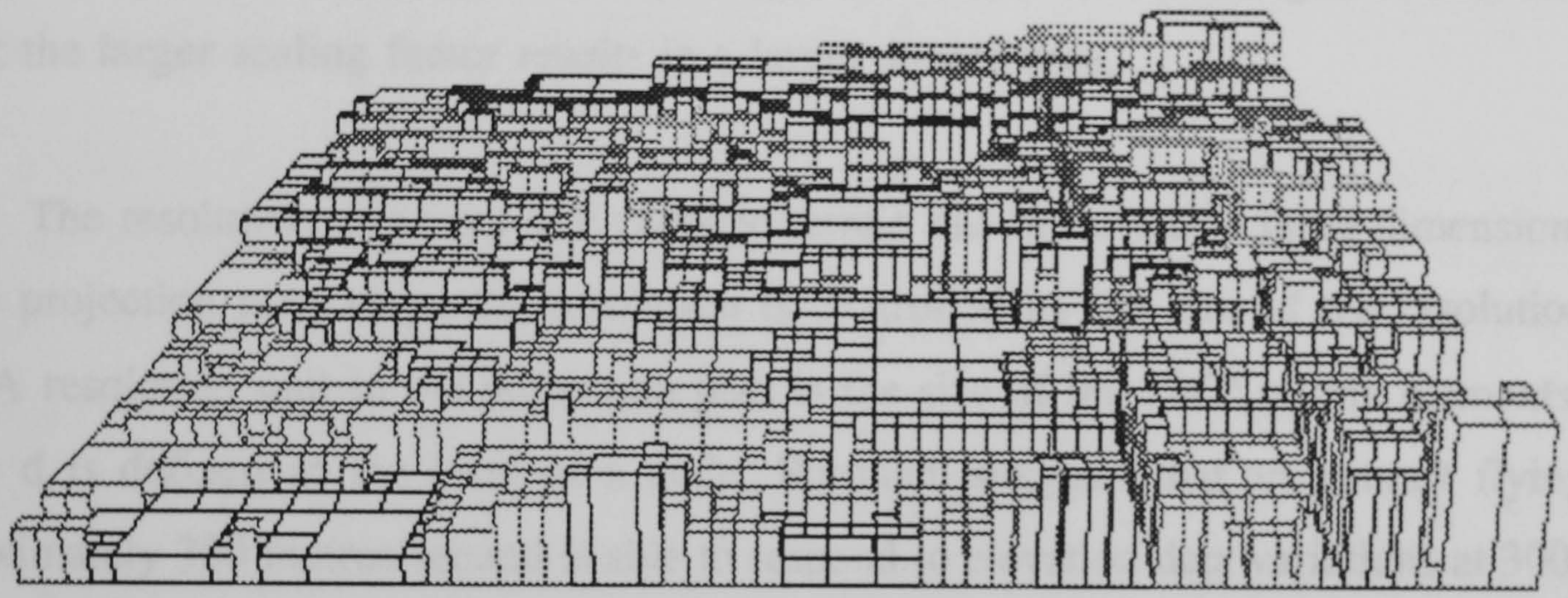


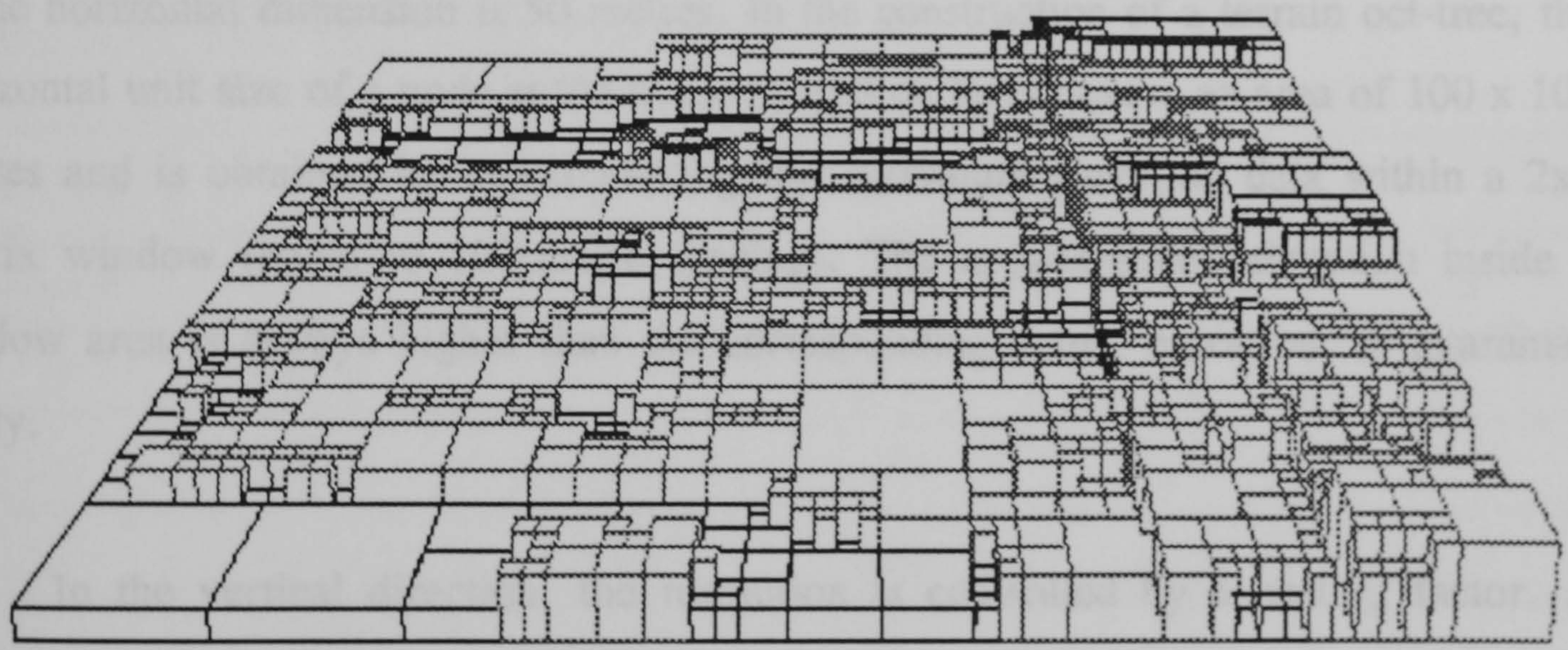
Figure 6.3 Comparison of linear and terrain oct-trees encoding methods using different scaling factors.



(a) A 64 x 64 grid file synthetic terrain .



(b) Oct-tree encoded terrain with a scaling factor 100 metres, 2584 nodes.



(c) Oct-tree encoded terrain with a scaling factor 200 metres, 1585 nodes.

Figure 6.4 An example of a 64 x 64 synthetic terrain with elevation range 0-800 metres in grid file (a) and terrain oct-tree (b,c) representations.

Figure 6.3 depicts the number of nodes in the oct-tree which almost decrease linearly with an increasing scaling factor for both encoding methods. The terrain oct-tree encoding method has achieved space efficiency by using the quadrants grouping scheme which increases the possibility of node merging instead of using the octants grouping method in which the merging can only apply to the nodes with odd scaled elevation value.

Figure 6.4 depicts a 64 x 64 synthetic terrain in grid-file co-ordinates and the equivalent oct-tree representation to illustrate the compression of oct-tree encoding, for scaling factors 100 and 200 metres respectively. For example, in the lower left corner of Figures 6.4b and 6.4c, the scaled homogeneous elements are merged to a $2^m x 2^m$ block; the larger scaling factor results in a larger square area.

The resolution of an oct-tree encoded terrain can be varied in three dimension. In the projection plan view, the resolution is controlled by the size of the resolution unit. A resolution unit in the projection plan is the size of the $2^d x 2^d$ matrix elements, where d is defined as the level of a node. Burnham suggests that an aircraft flying approximately 300 metres/second is able to respond to elevation data variations at 300-metres spacing [Burn84]. In this thesis, the original resolution of the DTM source files in the horizontal dimension is 50 metres. In the construction of a terrain oct-tree, the horizontal unit size of a node at the finest resolution level covers an area of 100 x 100 metres and is obtained by approximating the maximum elevation data within a 2x2 matrix window (equal to 100-metre spacing). The approximated elevation inside a window area is always higher than the corresponding actual elevation, to guarantee safety.

In the vertical direction, the resolution is controlled by a scaling factor. A DTED is converted to an oct-tree representation according to a given scaling factor, which divides the elevation space into horizontal bands. Non-linear scaling can be applied to the terrain elevation data in order to maximise the distribution of elevation data and to select the number of bands according to the accuracy of the application.

However, the distribution of the elevations in a DTED file needs to be examined to define a non-linear scaling scheme which will increase the cost of the encoding and decoding process.

Alternatively, if a reference elevation is defined as a 'baseline', the scaled elevation K of an element defines elevation as all elevation below the reference elevation. The scaling process is applied only to the elements with elevation values above the reference elevation. Before the elevation is divided by a scaling factor, the reference value is subtracted from the elevation. Elements with elevation less than reference elevation are assigned a scaled elevation $K=0$. The baseline scaling improves the distribution of resolution in the vertical direction and retains the nodes with critical elevation values.

Tables $6.2_{a,b}$ give two examples of the comparison of the terrain oct-trees in which the terrain oct-trees are derived from the same DTM file giving different baselines reference elevations. Accordingly, as indicated in Figure 6.5, the higher the baseline elevation, the smaller the number of nodes. It can be observed from Figure 6.5 that there is no significant reduction in the number of nodes when a larger scaling factor is used compared with a smaller scaling factor in a high baseline elevation. Thus, if the attention is focused on the detail of the high elevation area, a high baseline and small scaling factor encoding scheme should be used .

BaseLine Elevation (Metre)	Scaling Factor (Metre)				
	10	50	100	150	200
0	48166	25174	14824	9619	7468
100	33628	19270	6616	7498	5239
200	23776	12958	5239	4090	2530
300	12539	6271	2095	1621	913
400	5299	2206	910	157	4

TABLE 6.2a. Number of nodes with respect to different baseline elevations in the terrain oct-trees encoded from 2^9x 2^9 source DTM file around Port Talbot area.

Baseline Elevation (Metre)	Scaling Factor (Metre)				
	10	50	100	150	200
0	54994	27364	16717	11494	8800
100	51655	25591	15086	9910	7633
200	40456	20572	11596	7585	5026
300	28453	14059	7588	4840	3127
400	18034	7702	3565	1282	595

TABLE 6.2b. Number of nodes with respect to different baseline elevations in the terrain oct-trees encoded from 2^9 x 2^9 source DTM file around Peak District area.

When a high baseline elevation is used, the resultant oct-tree is similar to a binary representation of free space and obstacle regions of the navigation space defined by minimum flight altitude as depicted in Figure 6.6. For example, the baseline elevation can be used to determine the minimum flight altitude and the objects above baseline can be taken as the obstacles in the navigation space. In Figure 6.6(f), the navigation space is either free space or obstacle regions.

The approach discussed above provides an efficient method of representing the danger area by checking through the oct-tree and tagging the danger nodes according to a given minimum flight altitude. For instance, in Figure 6.7, a DTM file of the Peak District area is encoded with a 500 metre baseline and various scaling factors ranging from 10 to 100 metres. If the 500 metre baseline elevation is treated as the minimum flight altitude, the terrain oct-tree representation defines objects above 500 metres and with considerably less nodes in comparison with the oct-tree encoded at the zero baseline elevation in Figure 6.6 (mean sea level).

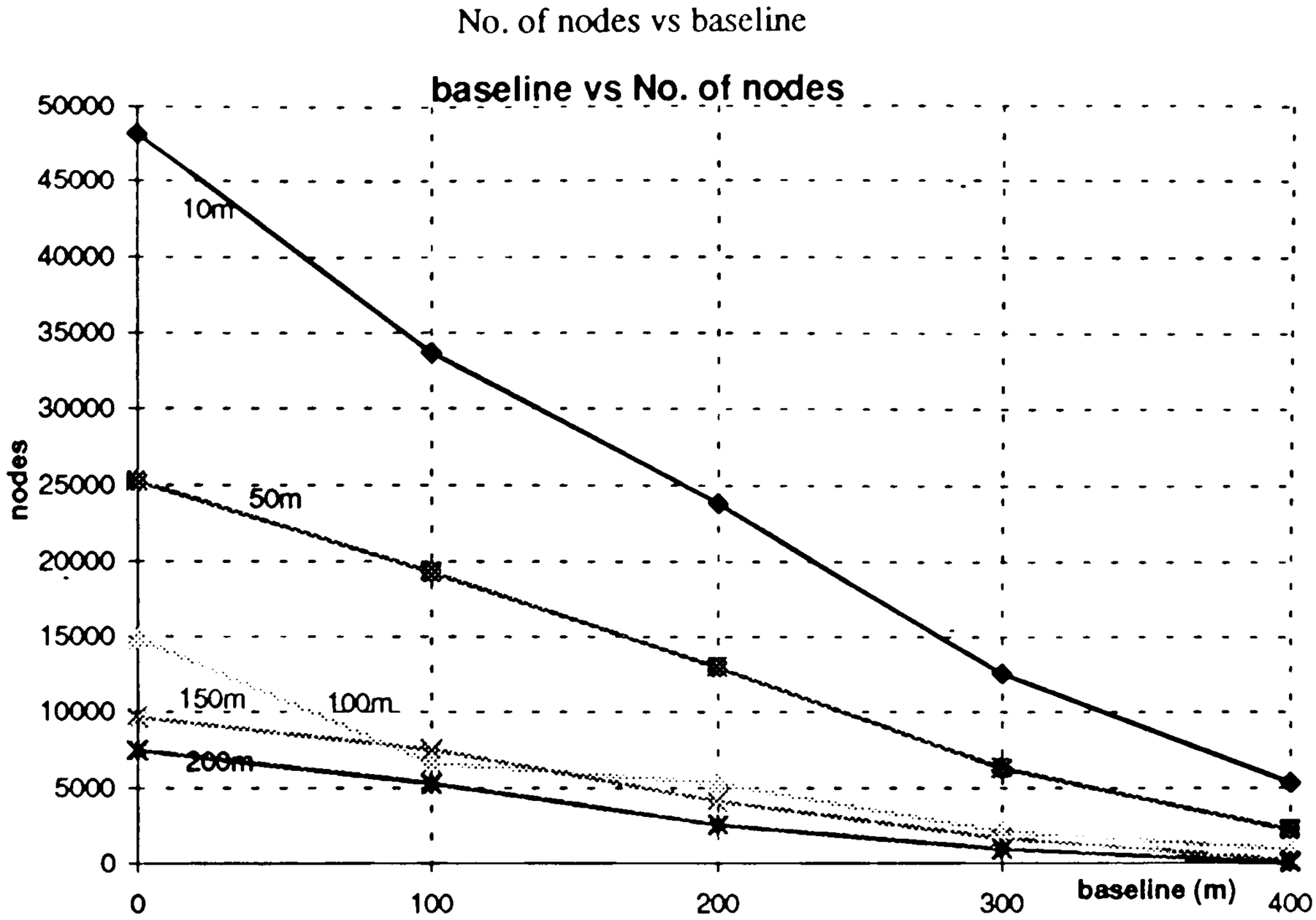


Figure 6.5a No. of nodes vs baseline elevation in the DTM file (Port Talbot area) encoded terrain oct-trees with different scaling factors.

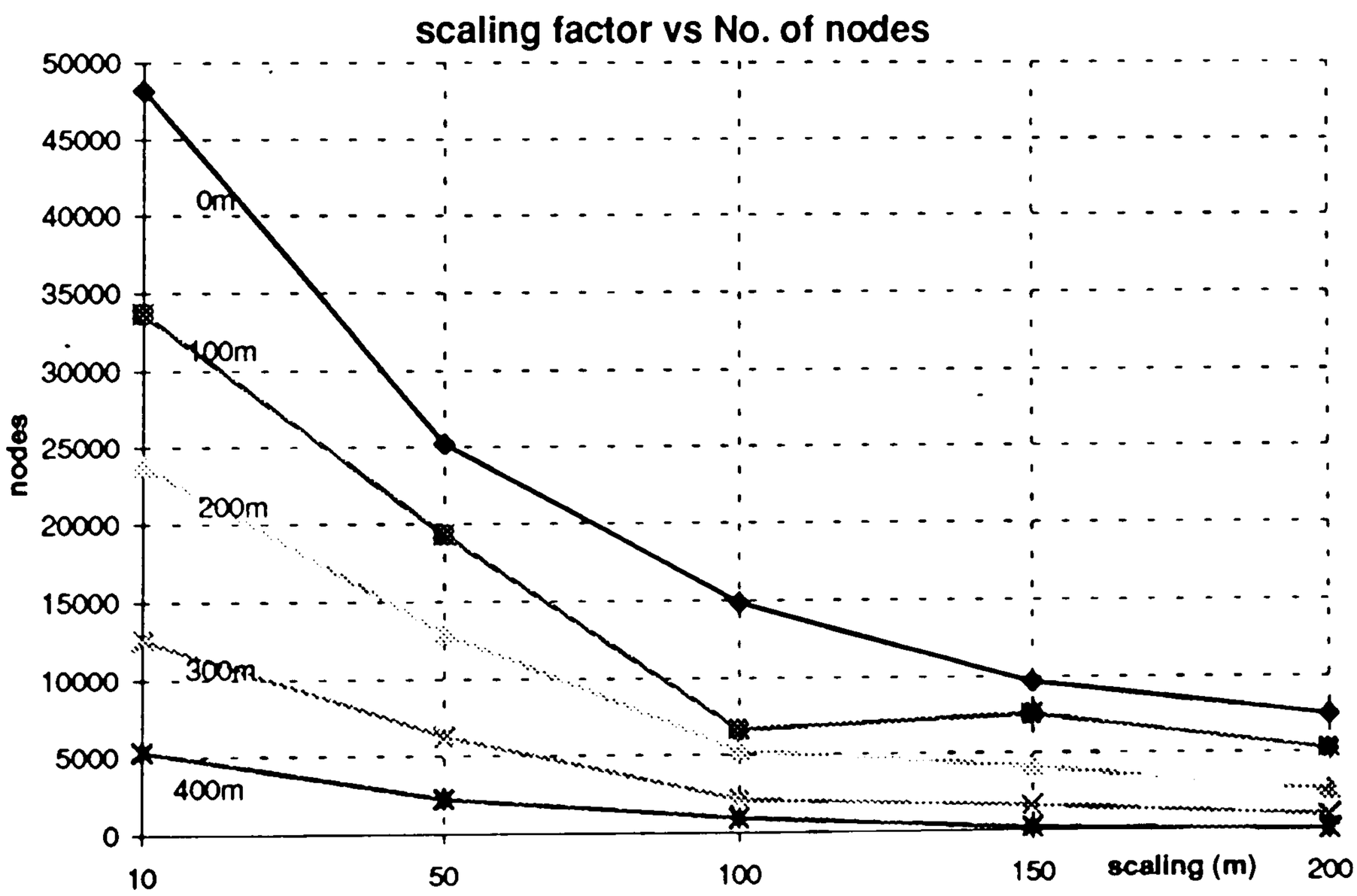


Figure 6.5b No. of nodes vs scaling factor in the DTM file (Port Talbot area) encoded terrain oct-trees based on different baseline elevations.

No. of nodes vs baseline

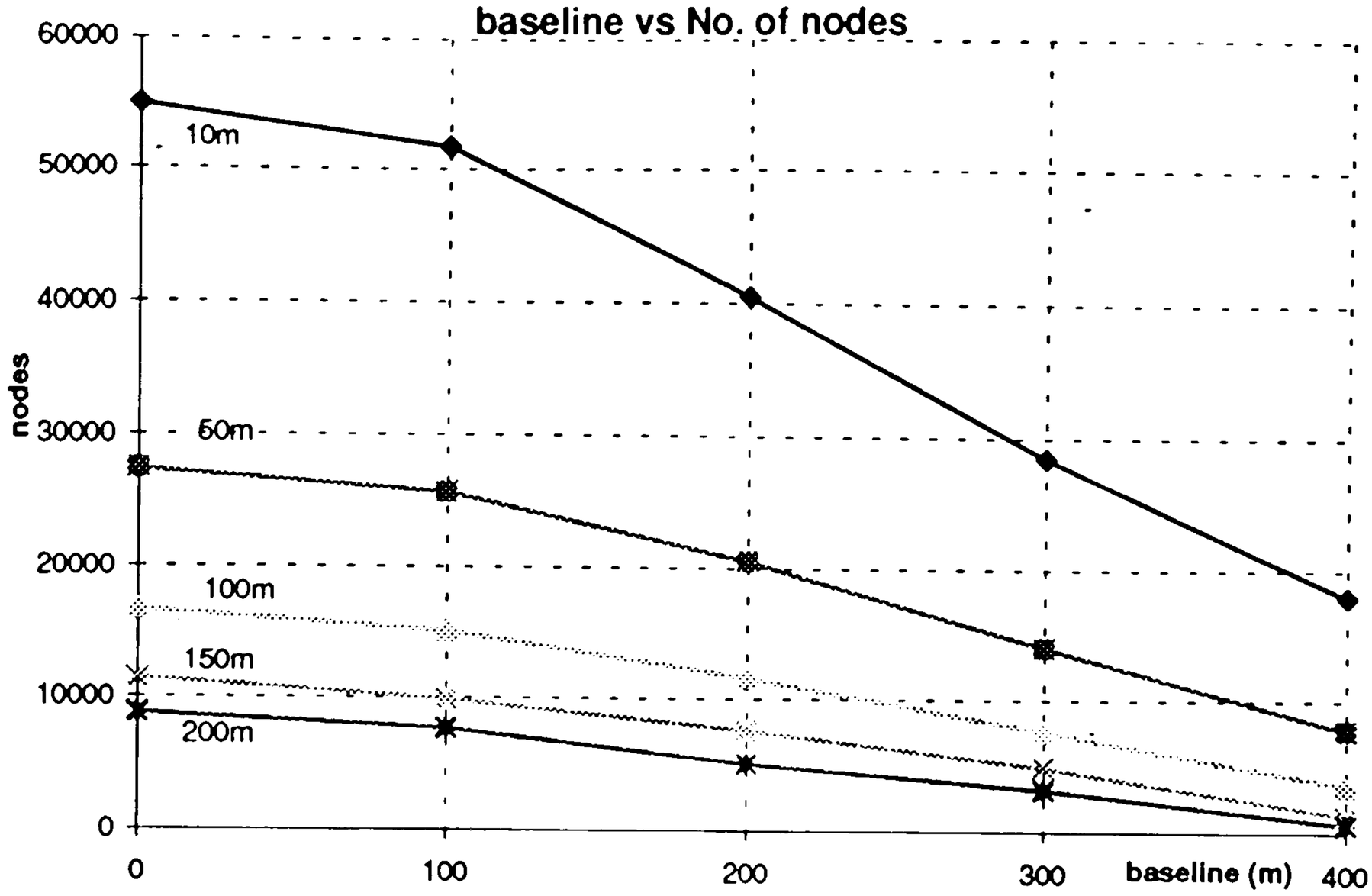


Figure 6.5c No. of nodes vs baseline elevation in the DTM file (Peak District area) encoded terrain oct-trees with different scaling factors.

No. of nodes vs scaling factor

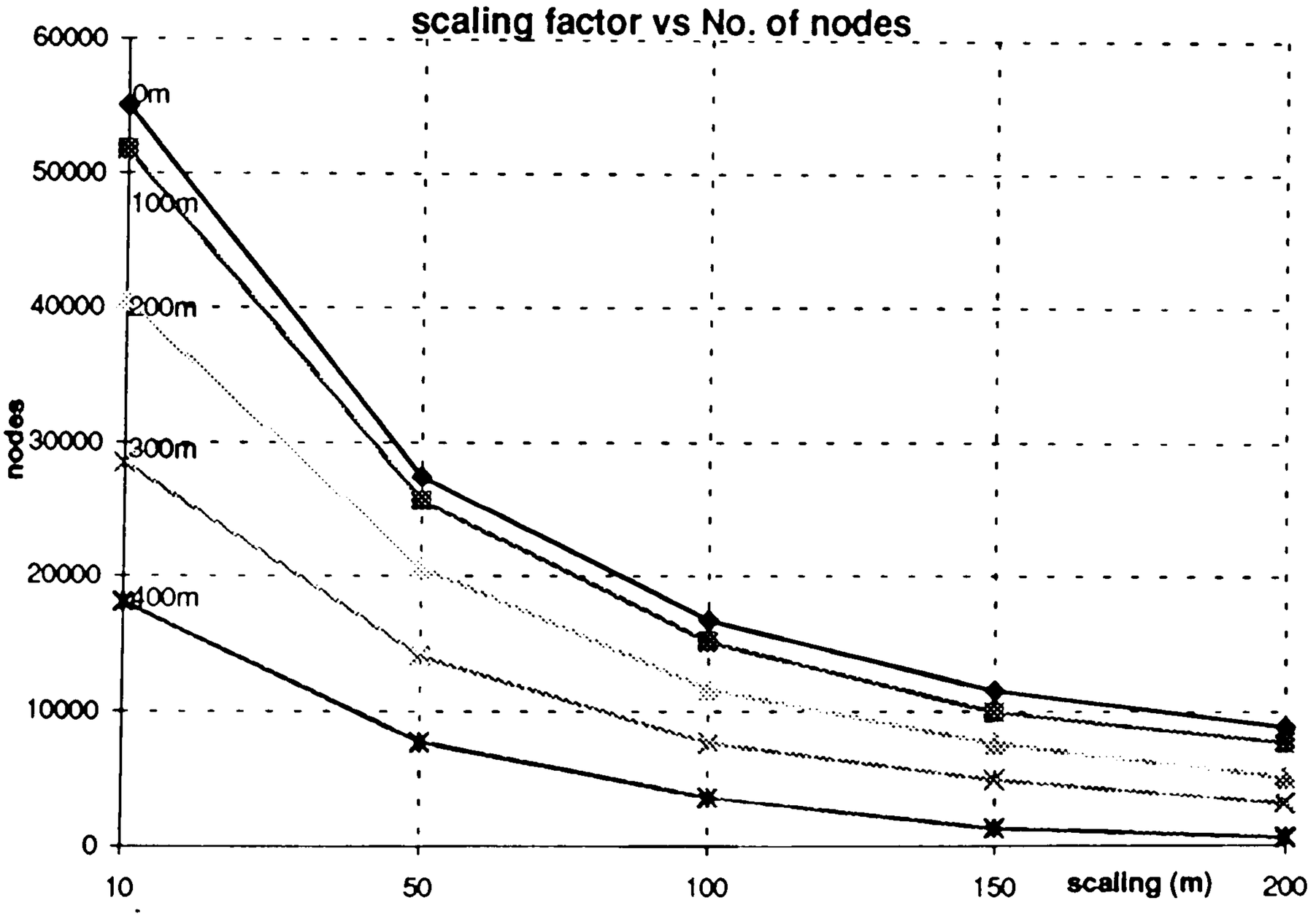
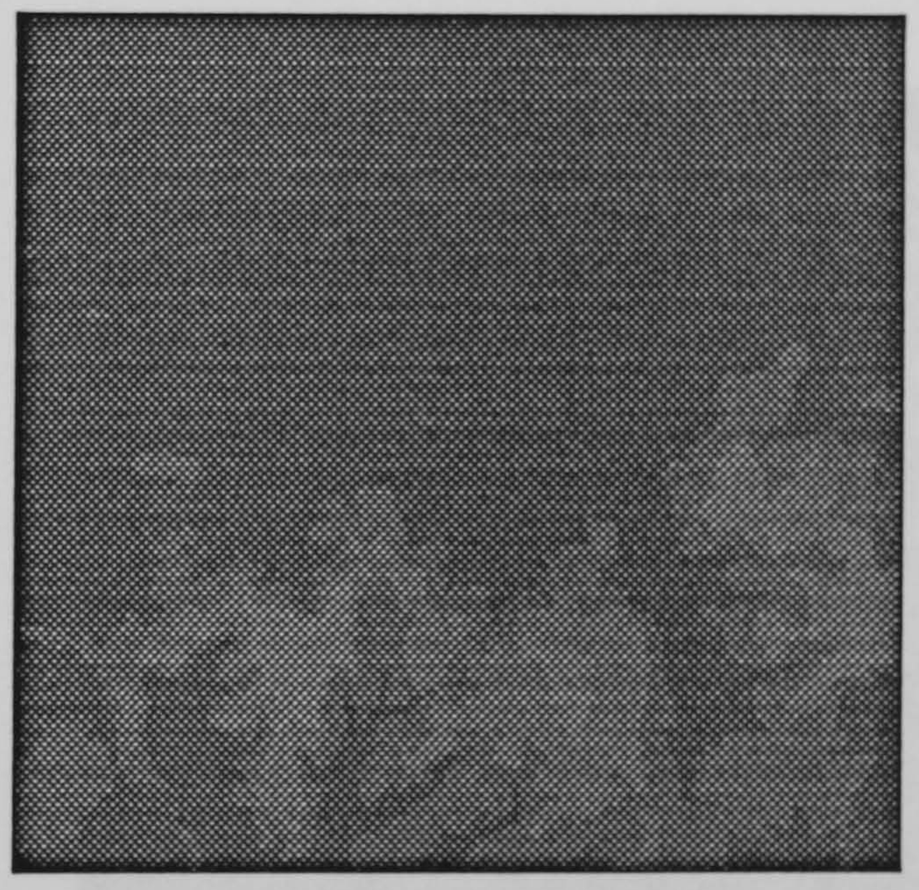


Figure 6.5d No. of nodes vs scaling factor in the DTM file (Peak District area) encoded terrain oct-trees based on different baseline elevations.

(a) Baseline elevation 0m, 16717 nodes.

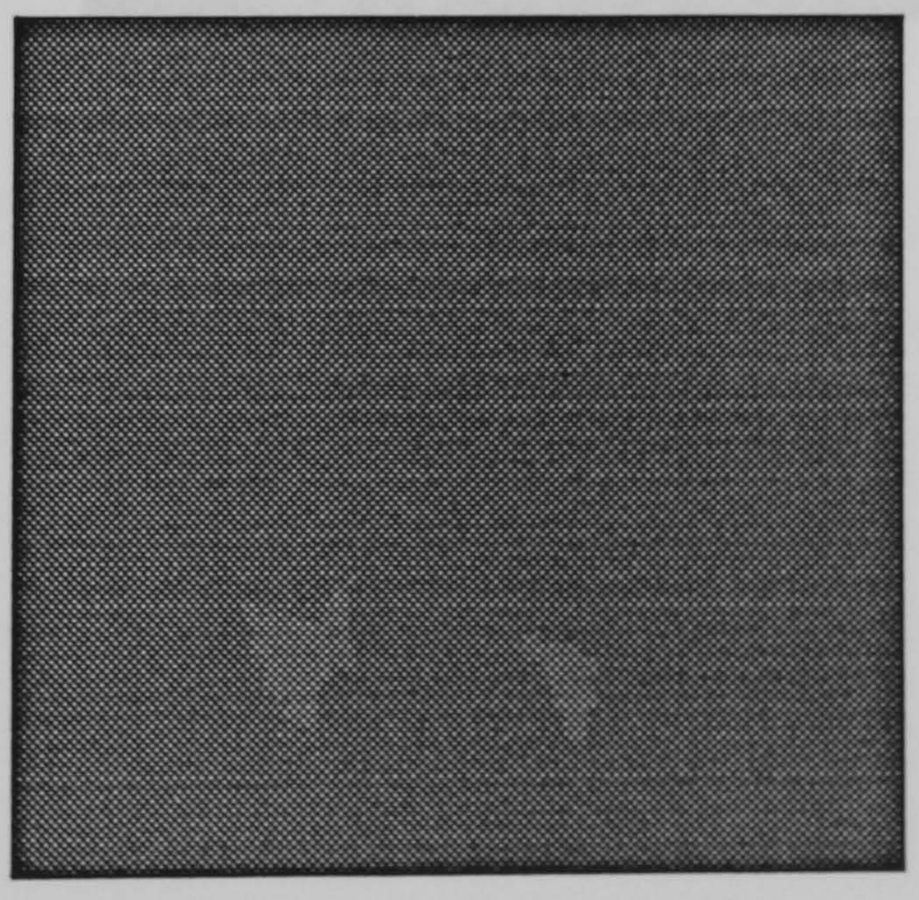
(b) Baseline elevation 100m, 15086 nodes.

(c) Baseline elevation 200m, 11596 nodes.



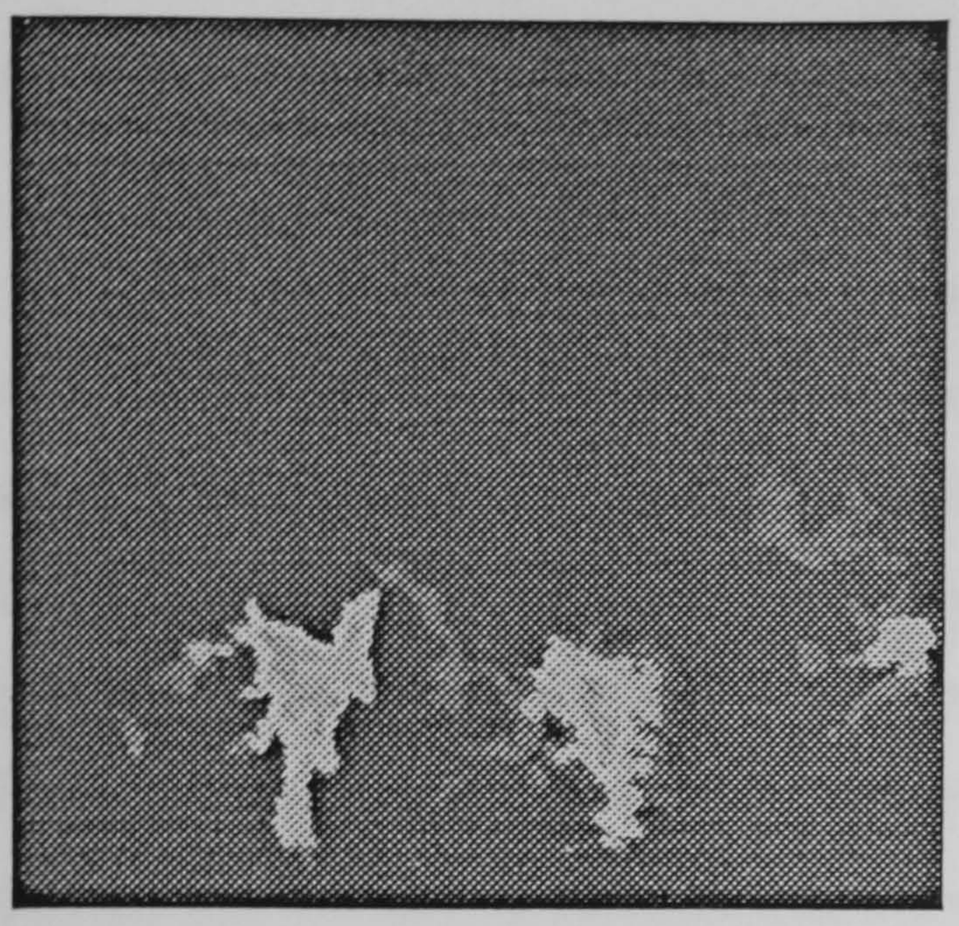
(d) Baseline elevation 300m, 7588 nodes.

(e) Baseline elevation 400m, 3565 nodes.



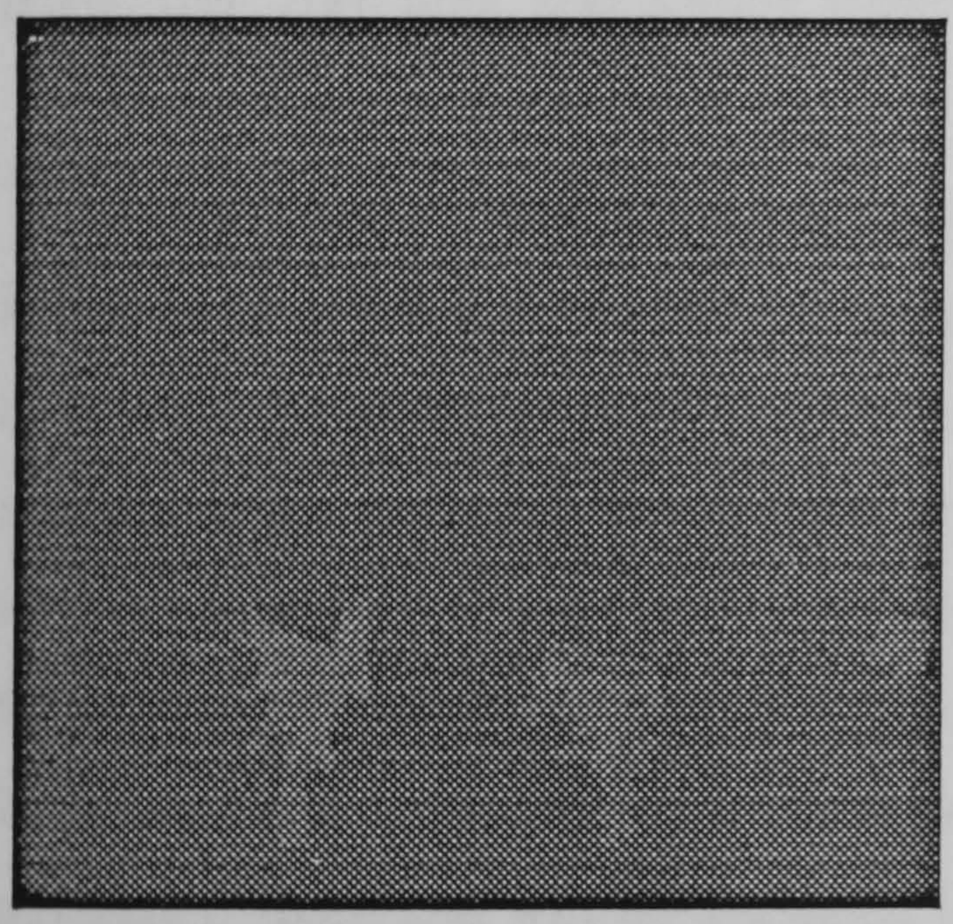
(f) Baseline elevation 500m, 595 nodes.

Figure 6.6 Examples of a 512 x 512 grid file (Peak district area) encoded terrain oct-tree representations using 100 metres as the scaling factor and different baseline elevations.

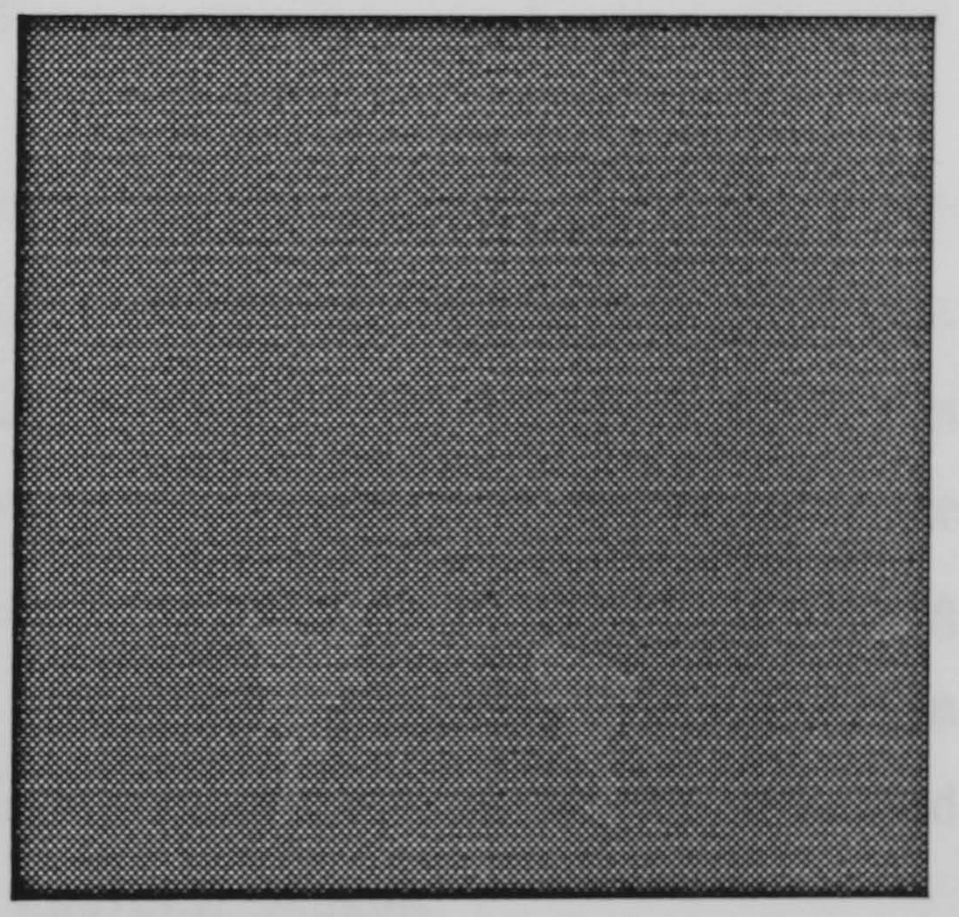


(a) Scaling factor 10m, 6436 nodes.

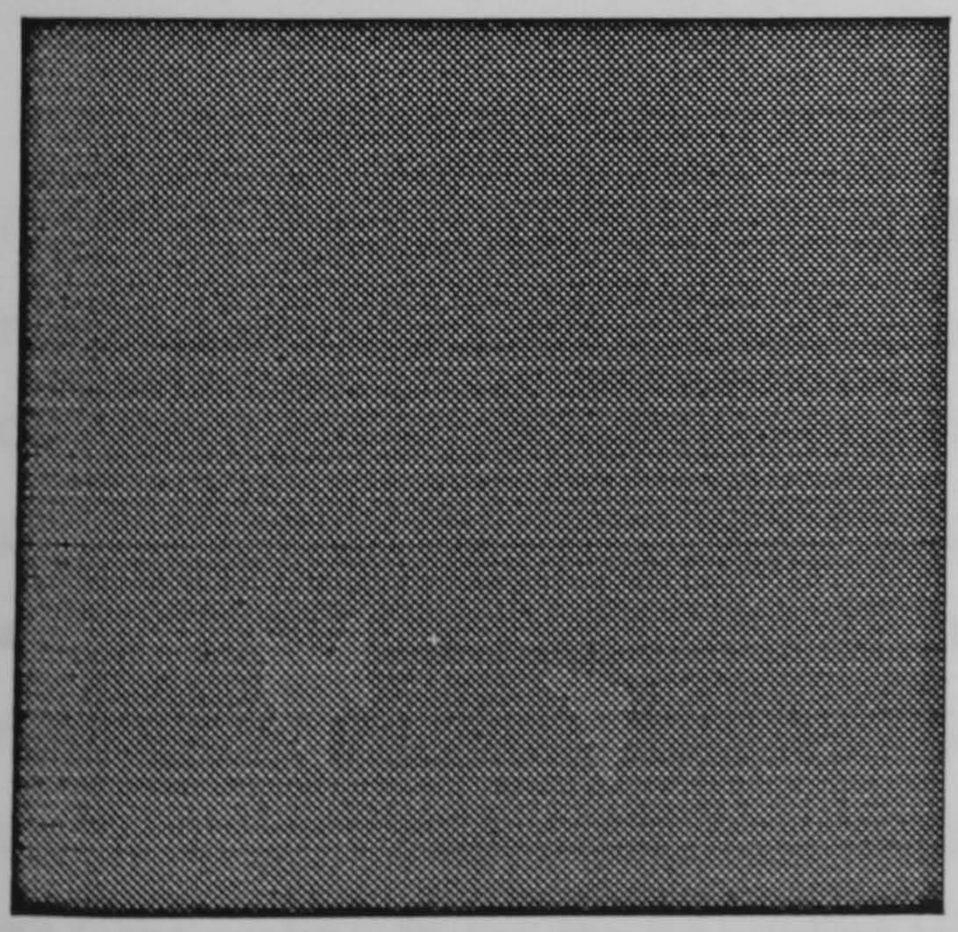
(b) Scaling factor 30m, 3130 nodes.



(c) Scaling factor 50m, 1642 nodes.



(d) Scaling factor 75m, 943 nodes.



(e) Scaling factor 100m, 595 nodes.

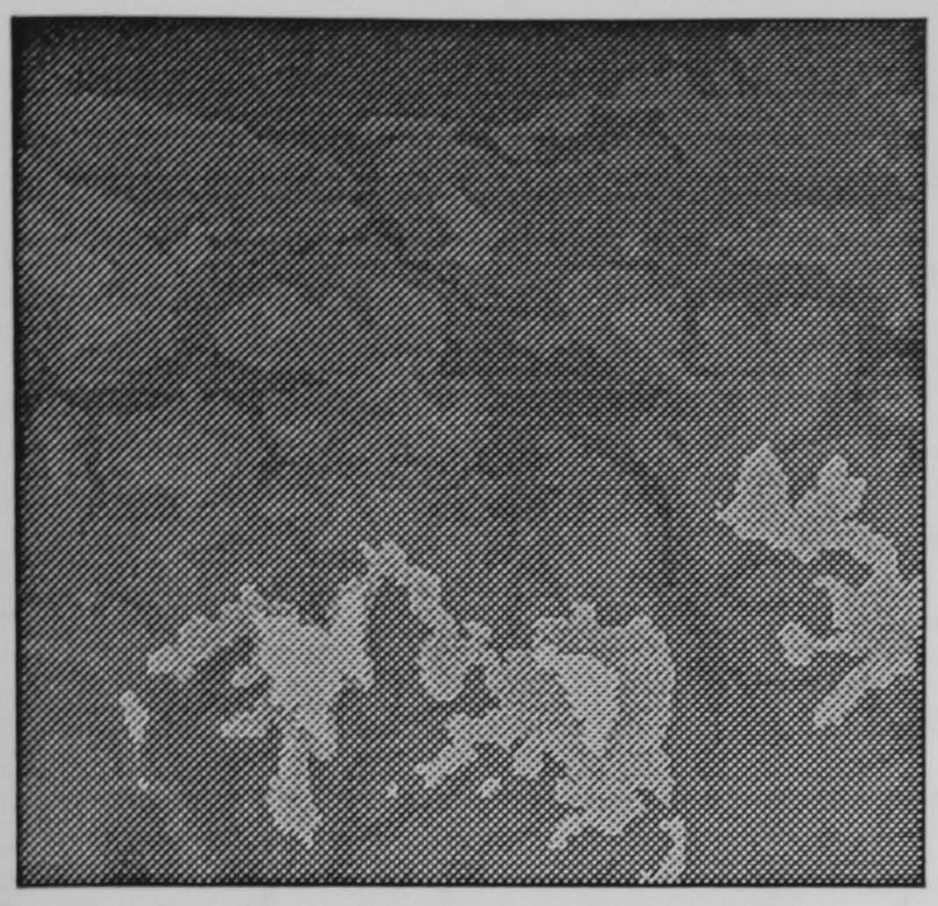
Figure 6.7 The comparison of 512 x 512 grid file (Peak District area) encoded terrain oct-tree representations using different scaling factors and a 500 metres baseline elevation.

6.2.2 Pyramids of Terrain Oct-trees

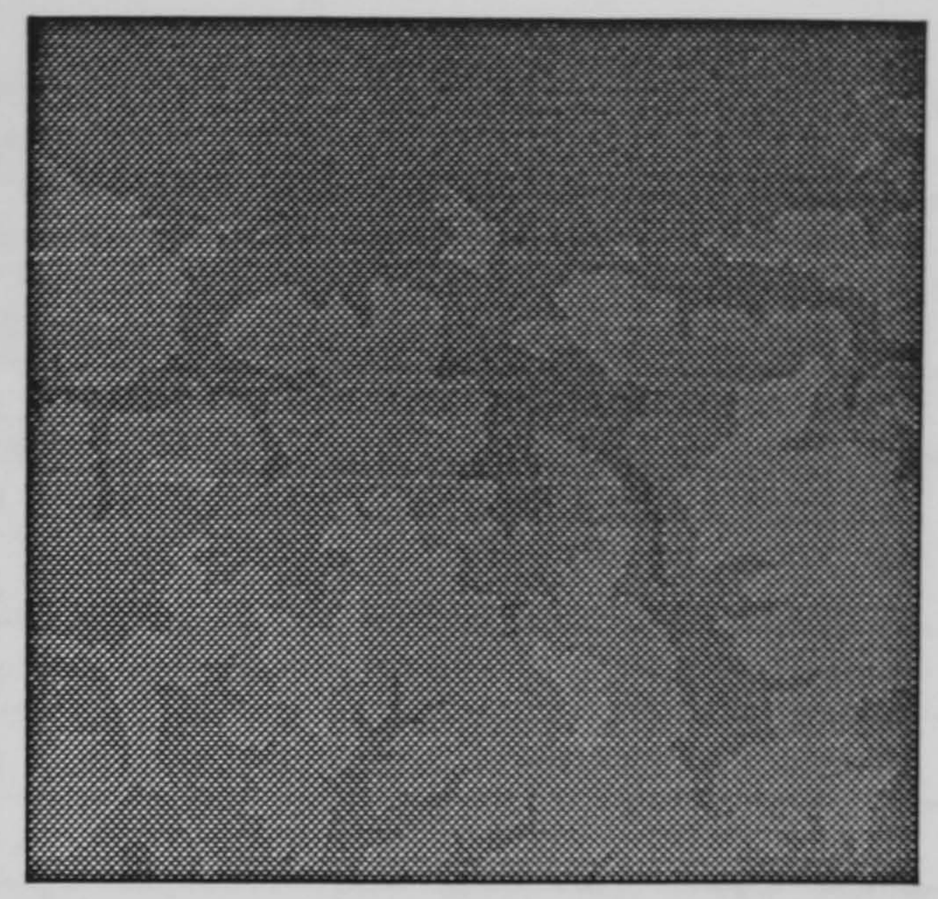
To access a pyramid of terrain oct-trees, the coarser layers in the pyramid are derived recursively from the higher resolution layer of a terrain oct-tree. Since an approximation function is applied to each 2 x 2 window area at the previous resolution layer, the number of nodes is further reduced. Each oct-tree in the upper layer of the pyramid has approximately (best case) a quarter of the number of nodes of its lower layer and retains the highest elevation of all of its sub-nodes. If a five-layer pyramid of a terrain oct-tree is constructed, the unit of resolution corresponding to each layer is 50, 100, 200, 400, 800 metres respectively.

Generally, a coarser scaling factor leads to fewer nodes, although the reduction is not necessary linear. For example, scaling factors above 300 metres do not lead to a significant reduction in the number of nodes. The results shows that the range of scaling factors applied to the encoding processing influences the number of nodes in the terrain oct-tree. Similarly for the baseline reference elevation case, if a large scaling factor is used, the terrain oct-tree approaches a binary image representation, as shown in Figure 6.8. The reduction to a binary image representation provides a simplified navigation space and will be discussed in the following sections.

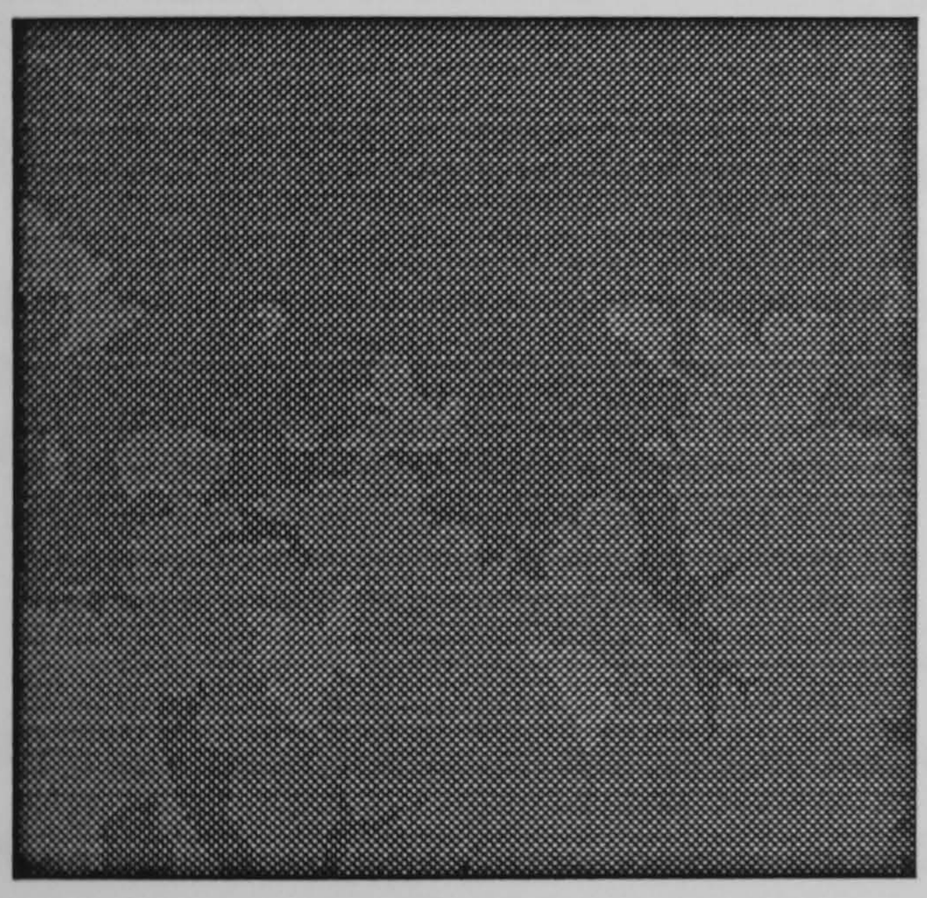
Both the scaling factors and the layers of the pyramid are used to reduce the number of nodes in a terrain oct-tree. The pyramid representation gives a substantial reduction in the number of nodes whereas the effect of using the scaling factor is terrain data dependent. Table 6.3 shows the number of nodes for the first six layers of a pyramid together with the reduction of data elements. By increasing the scaling factor at each layer of the pyramid, the results show a further reduction in the number of nodes which range from 27.7% (layer 6, scaling factor 50 metres) to 0.04% (layer 1, scaling factor 500 metres). The results imply that both the upper layer of a pyramid and the large scaling factor can be applied together to obtain an appropriate terrain oct-tree for a specific application. For example, the danger areas are highlighted in an upper layer of a pyramid with respect to flight distance according to a minimum flight altitude.



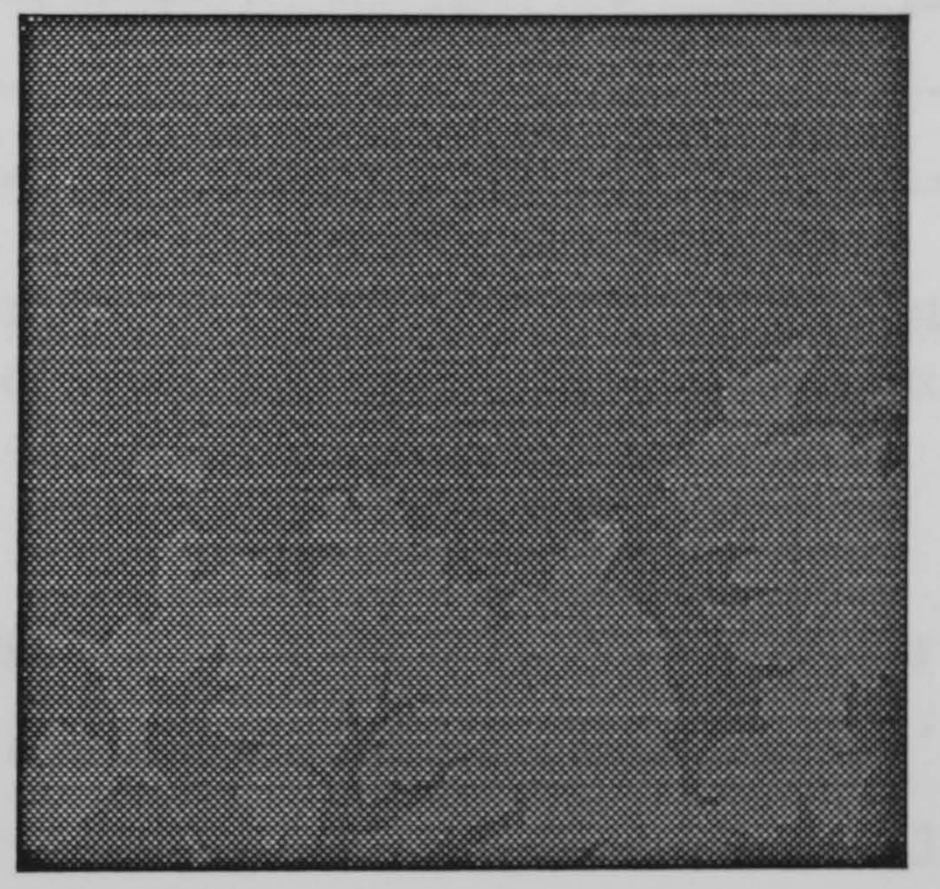
(a) Scaling factor 100m, 16717 nodes.



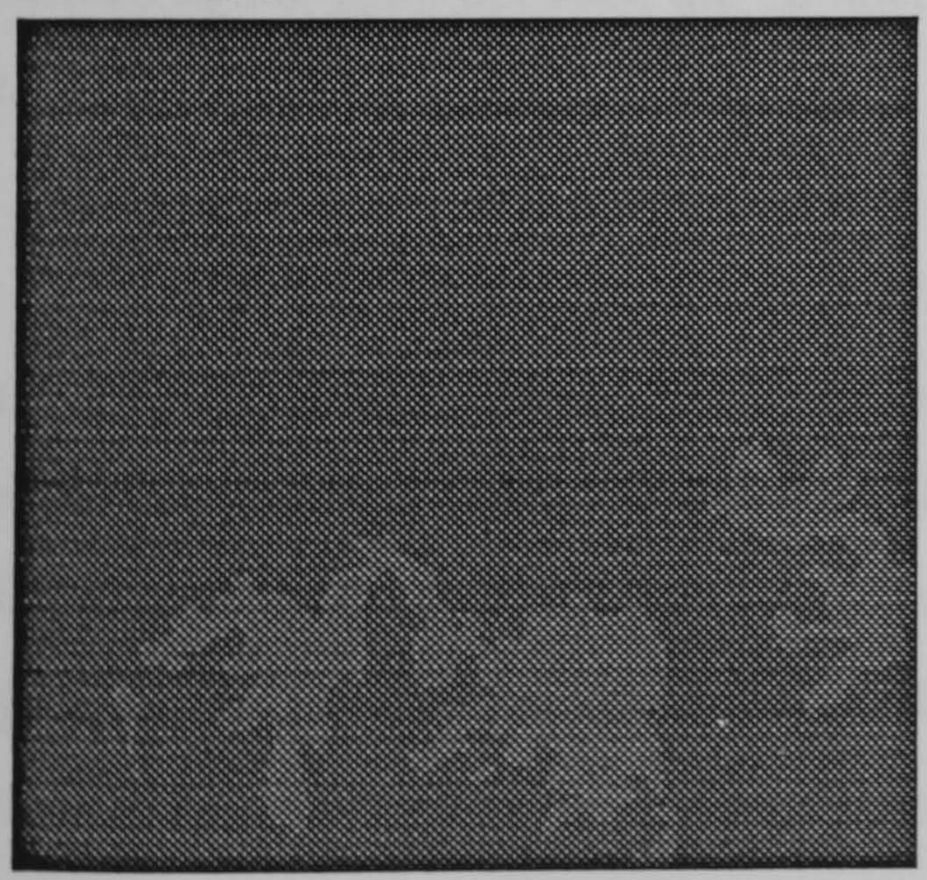
(b) Scaling factor 200m, 8800 nodes.



(c) Scaling factor 300m, 5125 nodes.



(d) Scaling factor 400m, 4504 nodes.



(e) Scaling factor 500m, 3127 nodes.

Figure 6.8 Examples of a 512 x 512 grid file (Peaks District area) encoded terrain oct-tree representations at layer two with different scaling factors.

Layer(size)	512*512 (6)		256*256 (4)		128*128 (4)		64*64 (3)		32*32 (2)		16*16 (1)	
Scaling(m)	nodes	%	nodes	%	nodes	%	nodes	%	nodes	%	nodes	%
50	72634	27.7	27364	10.4	12082	4.6	3733	1.4	1000	0.4	253	0.1
100	41388	15.8	16717	6.4	8659	3.3	3217	1.2	961	0.4	253	0.1
150	26457	10.1	11494	4.4	6229	2.4	2584	1.0	847	0.3	238	0.09
200	20884	8.0	8800	3.4	4924	1.9	2152	0.8	781	0.3	229	0.09
250	17504	6.7	7579	2.9	4267	1.6	1912	0.7	694	0.3	217	0.08
300	11827	4.5	5125	2.0	2914	1.1	1348	0.5	538	0.2	187	0.07
350	11427	4.4	4741	1.8	2731	1.0	1249	0.5	499	0.2	178	0.07
400	10954	4.2	4504	1.7	2560	1.0	1153	0.4	454	0.2	151	0.06
450	9694	3.7	4366	1.7	2461	0.9	1120	0.4	433	0.2	136	0.05
500	6835	2.6	3127	1.2	1786	0.7	881	0.3	310	0.1	109	0.04

TABLE 6.3. The number of nodes and the reduction of data items (No. of elements / No. of nodes) at the first six layers of a set of terrain oct-tree pyramids with respect to different scaling factors.

Based on the results in Table 6.3, Figure 6.9(a,b) illustrates that the number of nodes is decreased as the resolution unit is increased; the data *'reduction rate'* is increased as the scaling factor is increased, where the *'reduction rate'* in Table 6.3 is defined as the ratio between the number of nodes in the terrain oct-tree and the number of point elements in the corresponding source DTM file.

Figure 6.10 illustrates a 512 x 512 DTM file encoded terrain oct-tree pyramid at different resolution layers. The pyramid consists of five layers, numbered 5 to 1, where the scaling factor is fixed at 100 metres, each layer is a terrain oct-tree at different horizontal resolutions, and layer 5 is the most detailed (finest resolution) terrain. The terrain representation is colour shaded. Each colour code represents a specific scaled elevation value. This terrain representation provides a compact approximation of navigation space which is used in the path planning algorithms developed in this research programme. The pyramid structure can also be incorporated in navigation systems to overlay obstacles (red colour shaded) and symbolic map displays in a similar way as shown, in Figure 6.10.

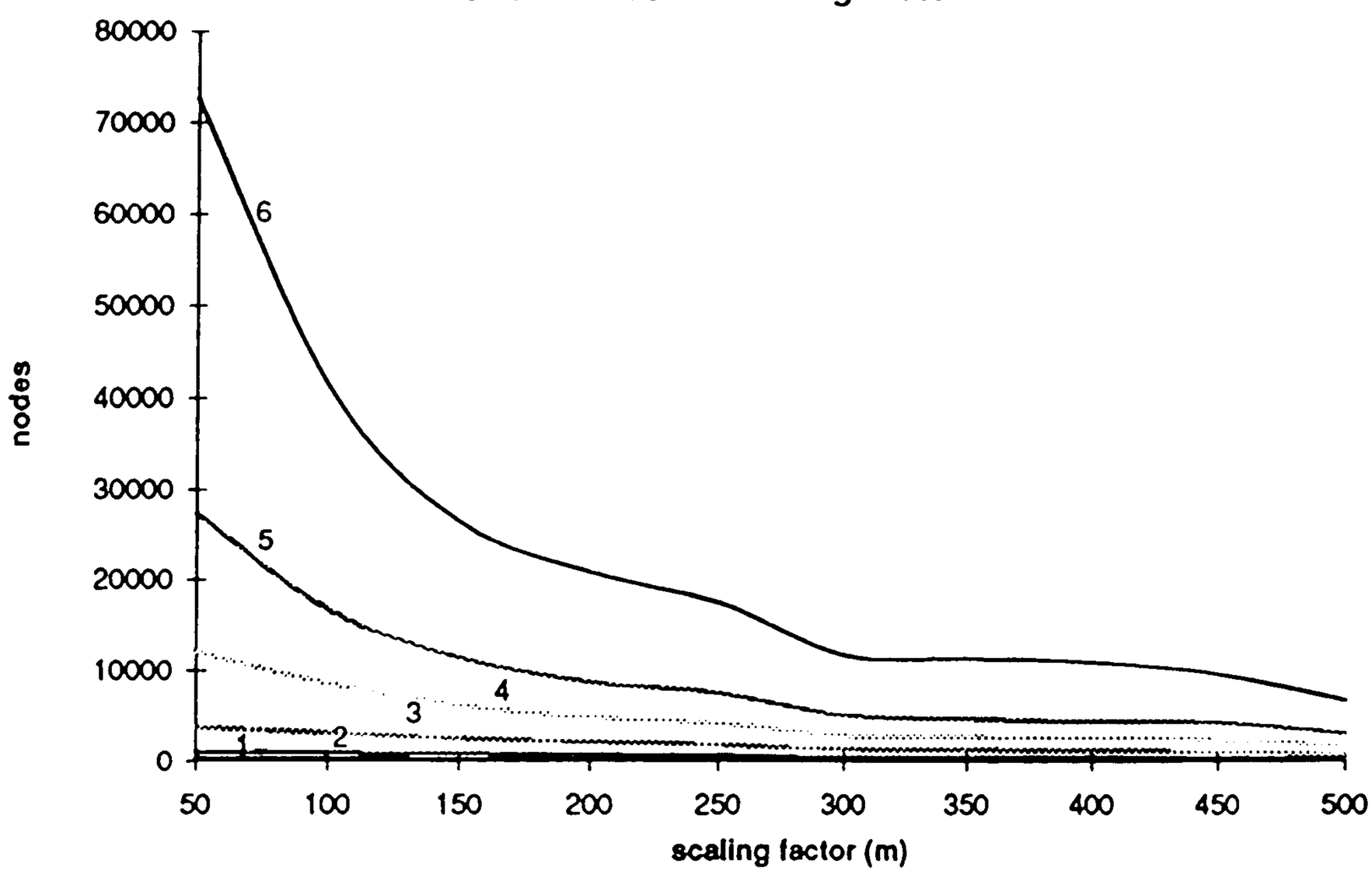
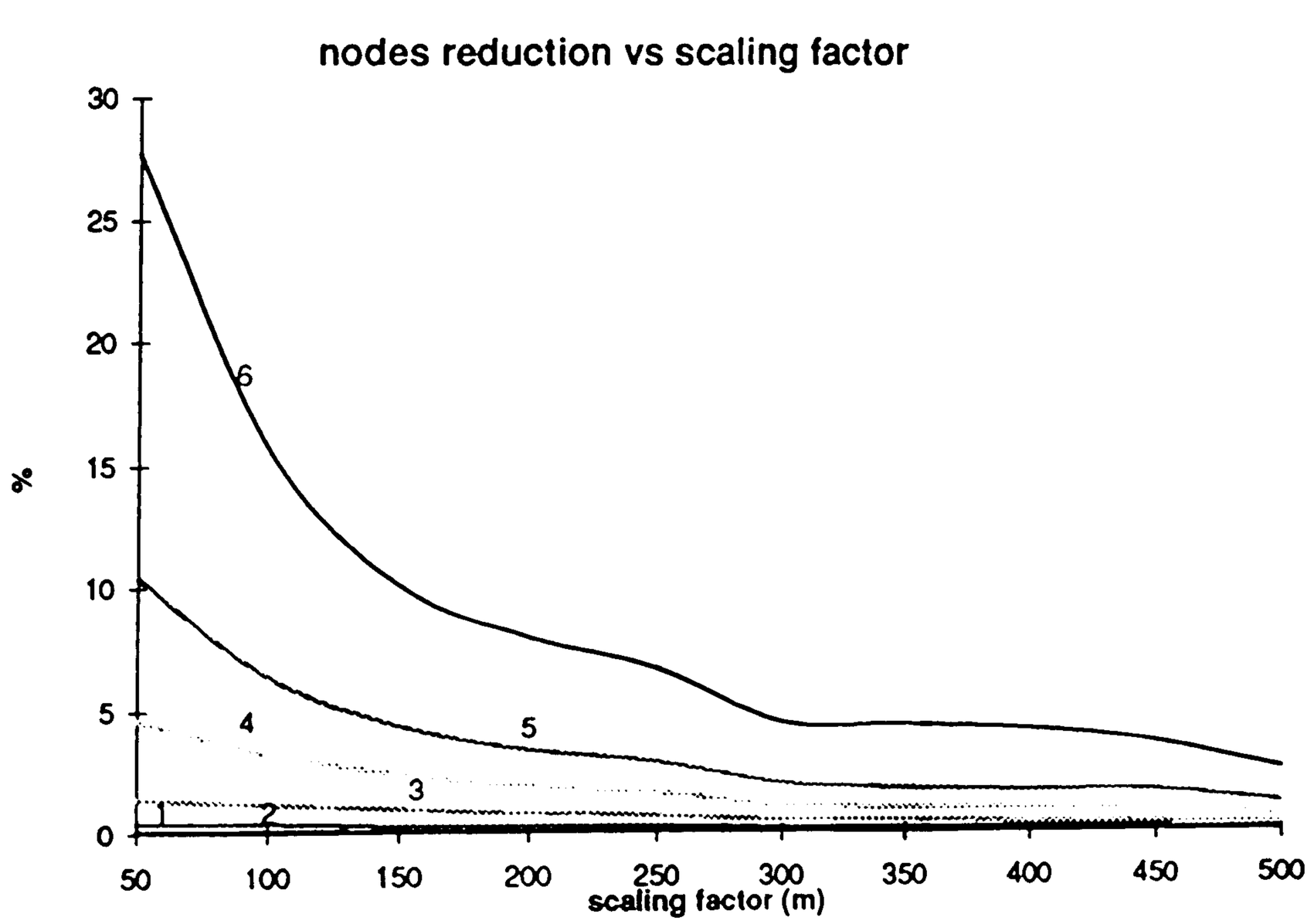


Figure 6.9a The number of nodes using different scaling factors.

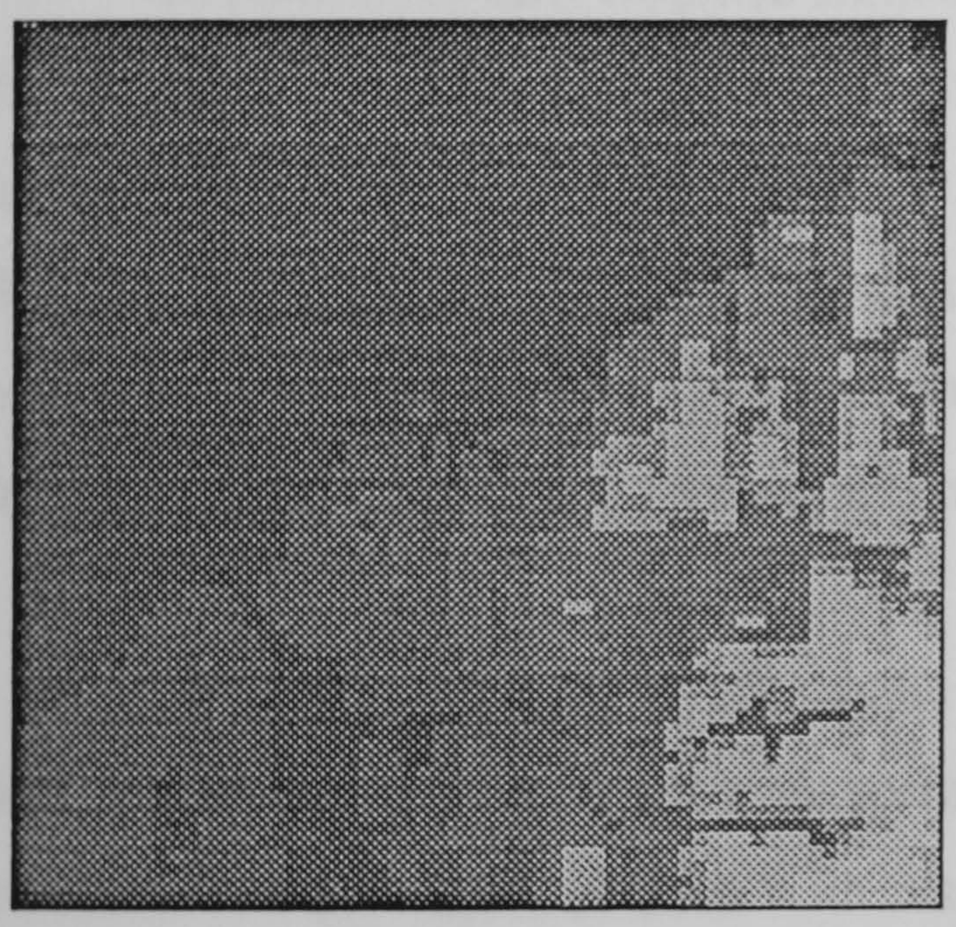


% ------ nodes/ No. of grid points

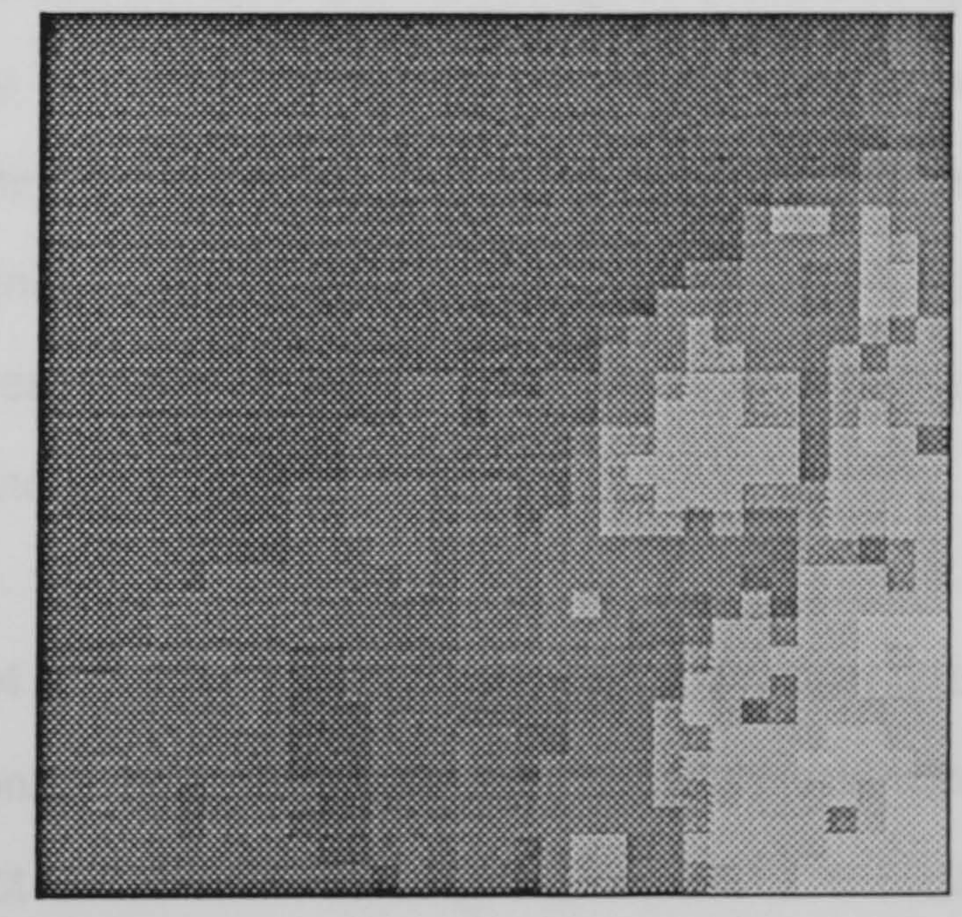
Figure 6.9b The nodes reduction using different scaling factors.

(a) Layer 5, 15475 nodes.

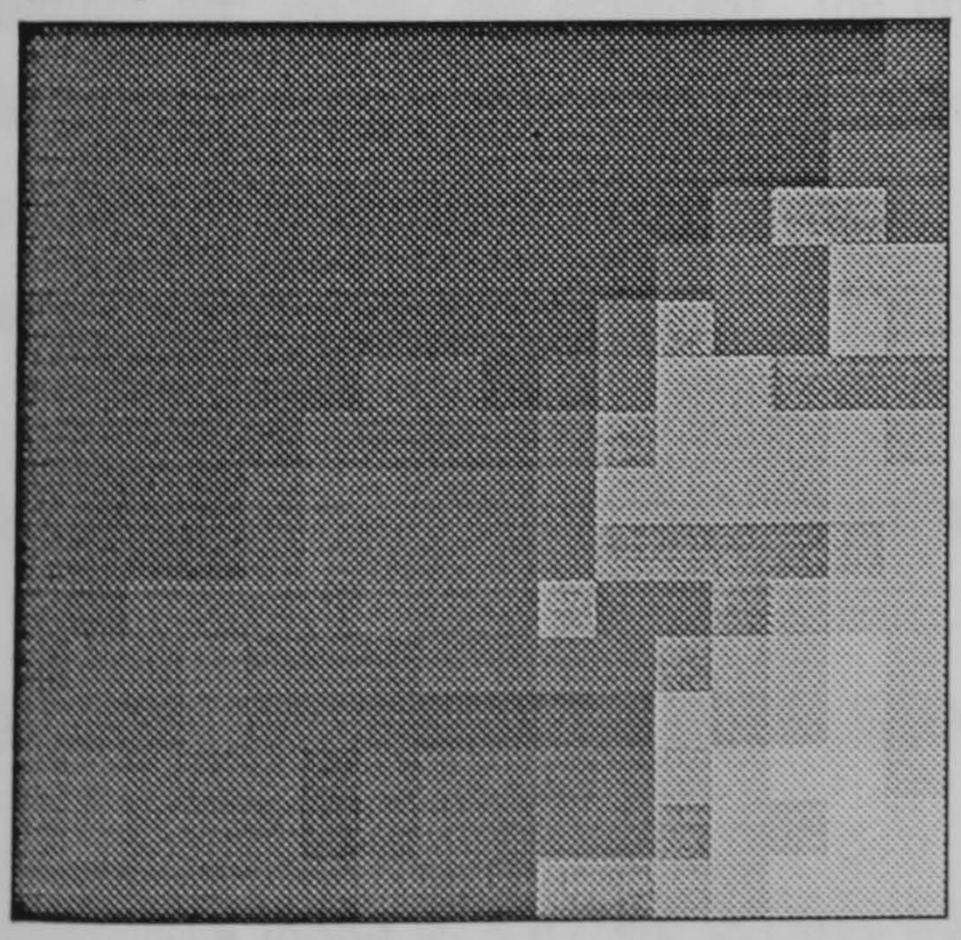
(b) Layer 4, 6619 nodes.



(c) Layer 3, 2101 nodes.



(d) Layer 2, 628 nodes.



(e) Layer 1, 178 nodes.

Figure 6.10 An example of 512 x 512 grid file (Port Talbot area) encoded terrain oct-tree in five layers pyramid representation with a fix scaling factor of 100 metres.

6.2.3 Elevation Accuracy in Terrain Oct-trees

When the elevation data is retrieved from the locational codes of a node, the use of a scaling factor introduces rounding errors into the elevation data stored in the oct-tree. The rounding error is bounded by the value of one scaling factor. For example, if a scaling factor of 50 is adopted for an encoding process, then elevations between 50 metres to 99 metres are rounded to the value 2 and the inaccuracy after retrieval is less than 50 metres.

The elevation data obtained from a terrain oct-tree is always higher (rounded up) than the true elevation to ensure that safety (in terms of terrain elevation during the navigation) can be guaranteed. However, the true elevation can be retrieved from the database using the locational codes as an index. As a node is encoded with the co-ordinates of its pixel element in the north-west corner, the horizontal error in a terrain oct-tree corresponds to the error figure of the original DTM file.

To verify the error figure, Table 6.4 provides the position and elevation data obtained in terrain oct-tree retrieval operations. The DTED elevation data, the rounded elevation in the terrain oct-trees with respect to different scaling factors and the error figures are given. As expected, the error is within scaling factor bounds. In the following sections, the interim results of the major stages of the static flight path planning algorithm are demonstrated. The time performance observations of the algorithm are also discussed.

6.3 The Results of Flight Path Planning Stages

6.3.1 The Modelling of Navigation Space

The nodes of a terrain oct-tree with an elevation value above a minimum flight altitude are known as "*danger nodes*", in the sense that an aircraft cannot safely enter a region of the terrain occupied by these nodes. Figure 6.11 shows an example of 3-D

graphic display of a 64 x 64 terrain grid file with an elevation range 0-800 metres. The *danger nodes* are highlighted from the encoded terrain oct-tree according to a minimum flight altitude of 500 metres and a scaling factor of 150 metres. Table 6.5 gives examples of the number of danger nodes with respect to the minimum flight altitude of two DTM encoded terrain oct-tree pyramids. It is clear that the number of danger nodes decreases as the flight altitude increases.

position (x,y)	Elevation DTED(m)	Oct-tree scaling 30m	disparity error (m)	Oct-tree scaling 50m	disparity error (m)	Oct-tree scaling 100m	disparity error (m)	Oct-tree scaling 150m	disparity error (m)
(65,190)	55	60	5	100	45	100	45	150	95
(85,170)	120	150	30	150	30	200	80	150	30
(115,140)	175	180	5	200	25	200	25	300	125
(135,120)	121	150	29	150	29	200	79	150	29
(145,110)	193	210	17	200	7	200	7	300	107
(165,90)	255	270	15	300	45	300	45	300	45
(175,80)	179	180	1	200	21	200	21	300	121
(195,60)	178	180	2	200	22	200	22	300	122
(205,50)	243	270	27	250	7	300	57	300	57
(215,40)	171	180	9	200	29	200	29	300	129
(225,30)	167	180	13	200	33	200	33	300	133
(235,20)	41	60	19	50	9	100	59	150	109

TABLE 6.4. The comparison of DETD elevation values with the elevation data retrieved from different scaling factor encoded terrain oct-trees.

Minimum Flight Altitude (m)	Number of nodes (Peak District)				
	Layer 5	Layer 4	Layer 3	Layer 2	Layer 1
200	13189	7036	2269	764	207
300	9318	5092	1898	584	165
400	5335	3128	1262	412	125
500	1934	1244	563	203	70
600	250	182	78	37	16

(Table 6.5a)

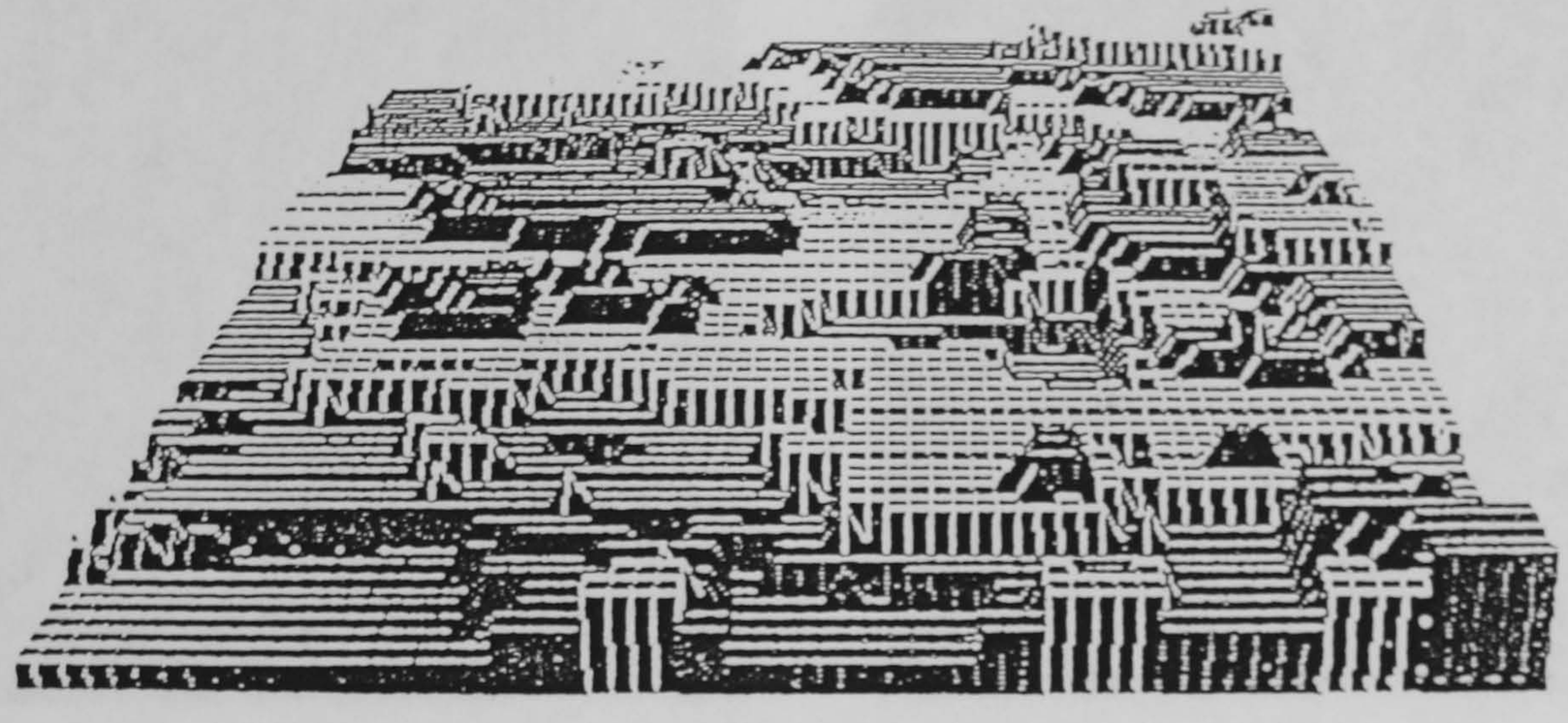
Minimum Flight Altitude (m)	Number of Nodes (Port Talbot)				
	Layer 5	Layer 4	Layer 3	Layer 2	Layer 1
100	12933	6635	2284	641	180
200	8991	4742	1684	488	136
300	4793	2629	970	303	94
400	1841	1093	432	148	49
500	978	260	114	48	19

(Table 6.5b)

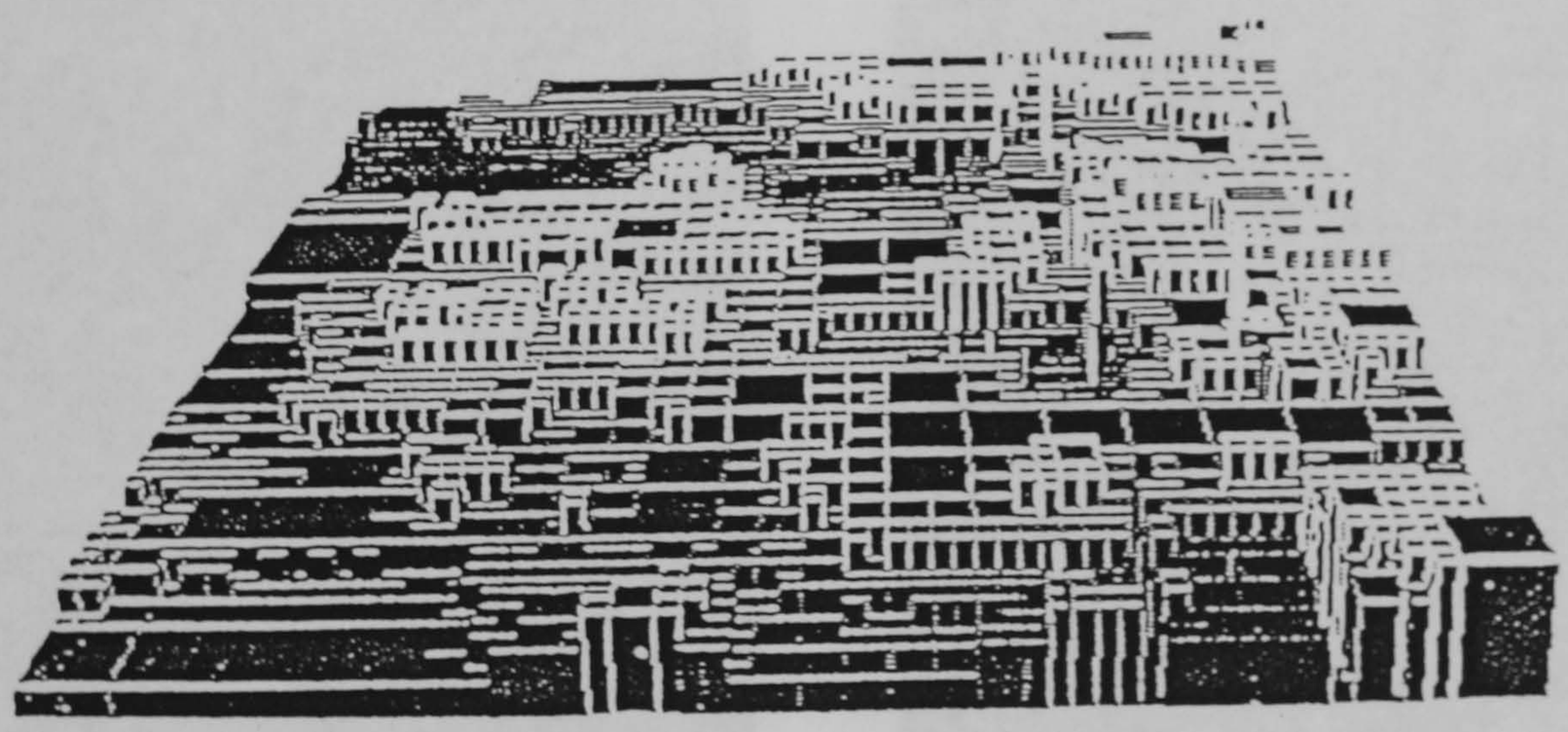
TABLE $6.5_{a,b}$. No. of danger nodes at different layers of DTM encoded terrain pyramid.

The danger nodes related to the current flight path direction which represent the coverage of obstacle regions are obtained by *collision checking* of a direct path between a given *start* point (*S*) and a *goal* point (*G*). The direct path is obtained by applying Bresenham's line generation algorithm between *S* and *G* to determine the oct-tree nodes on a straight line between them. The list of leaf nodes along the straight line which has been identified in collision with the danger area is expanded to form the obstacle regions.

The node element along a line is the size of a resolution unit, thus the number of nodes is proportional to the length of the line segment and the resolution level. Figures 6.12 and 6.13 depict the two-dimensional representations of 512 x 512 DTM file encoded terrain oct-trees, the Bresenham's line segment representation between the start and goal points and the intersected obstacle regions at different resolution layers.



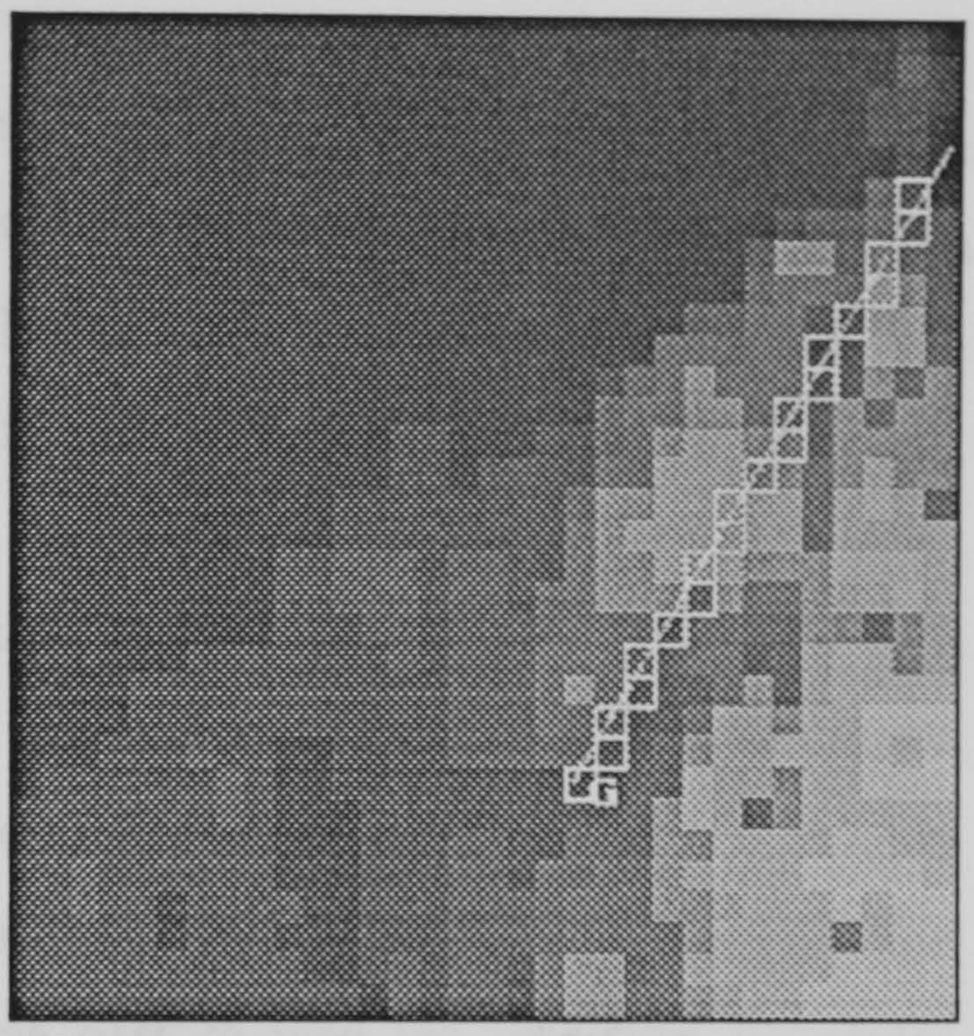
(a) A 64x64 terrain grid file with elevation range 0-800 metres.



(b) Terrain oct-tree highlighting elevation above 500 metres..

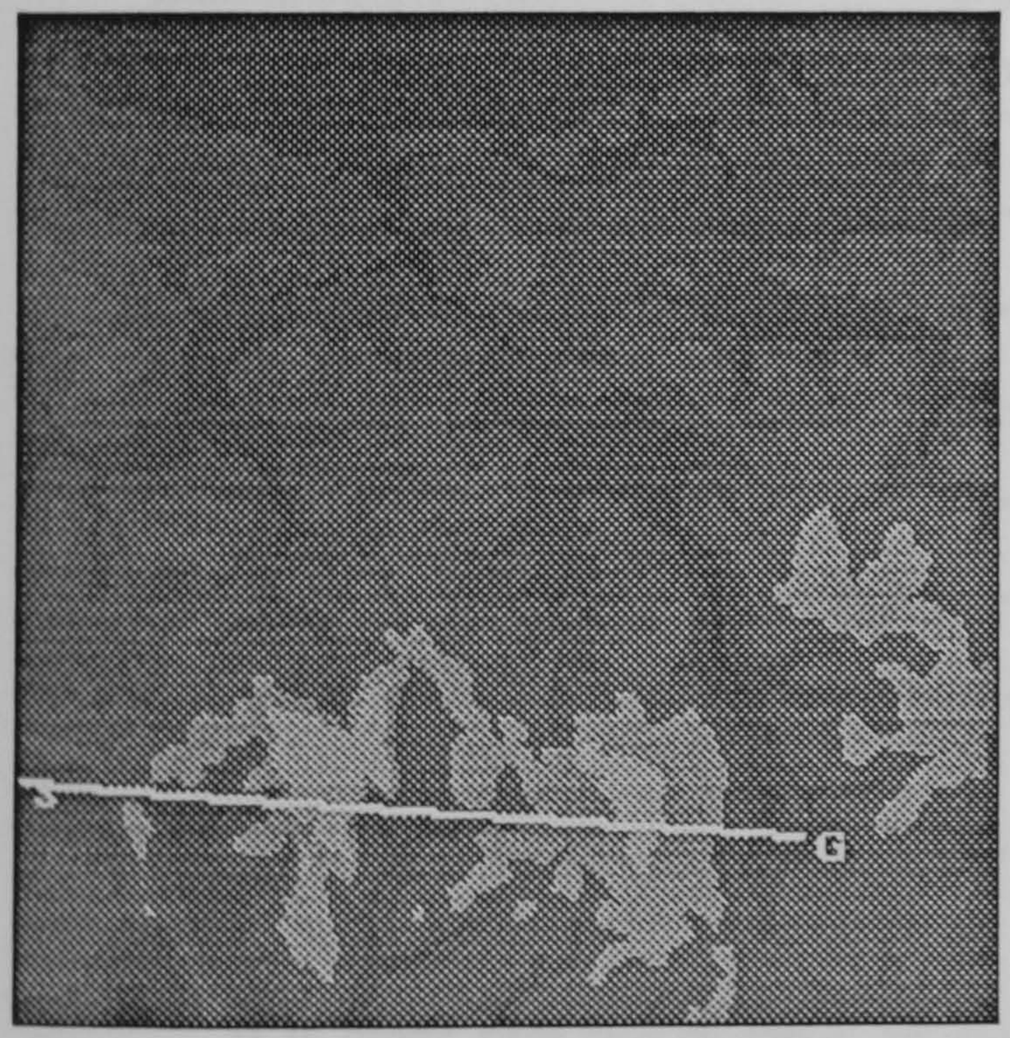
Figure 6.11 A terrain oct-tree with danger nodes highlighted.

1242 danger nodes .

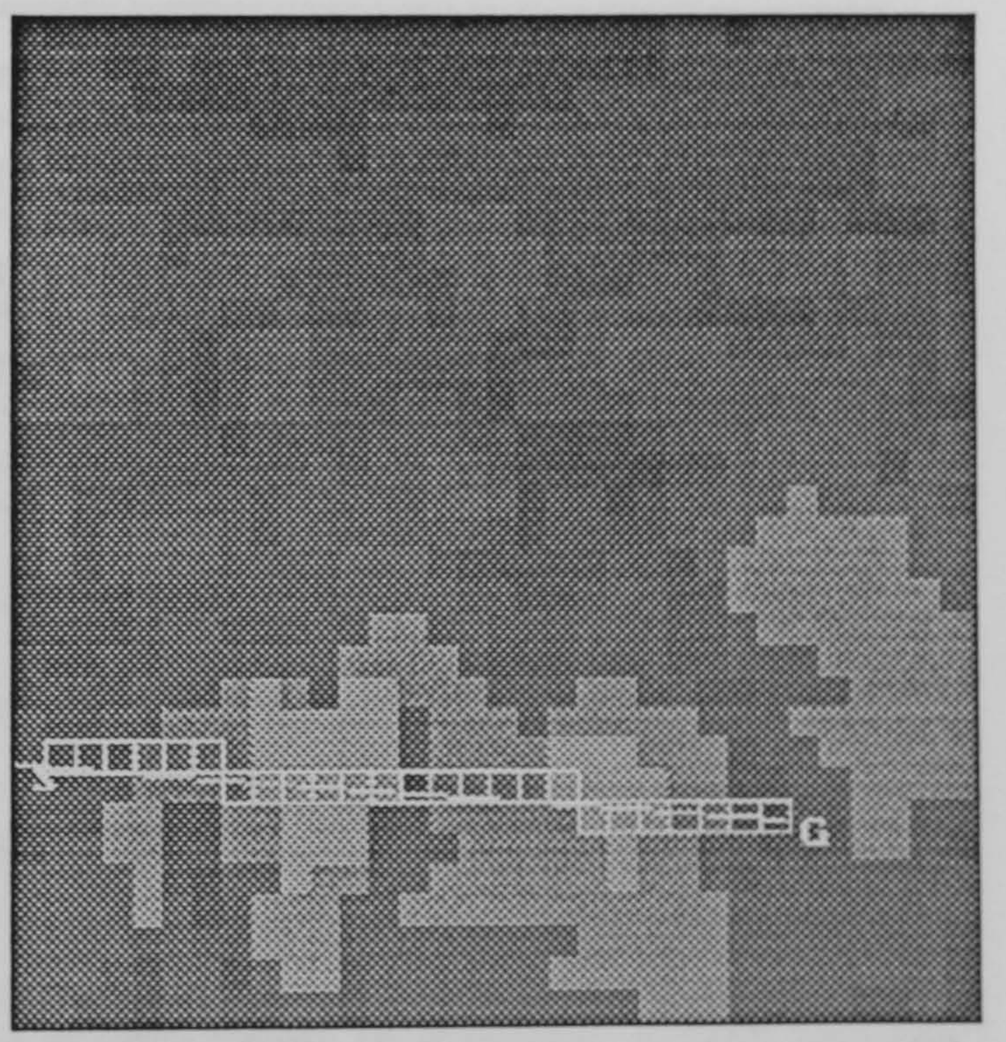


129 danger nodes.

Figure 6.12 Examples of obstacle regions and Bresenham's lines at layer 2 and 4 result from obstacles expansion process. The minimum flight altitude is 600 metres.



250 danger nodes.



37 danger nodes.

Figure 6.13 Examples of obstacle regions and Bresenham's lines at layer 1 and 4 result from obstacles expansion process. The minimum flight altitude is 600 metres.

6.3.2 2-D Locational Codes as Indices

In this section, the nodes along a line generated by Bresenham's algorithm are used to demonstrate efficient access operations for a terrain oct-tree. Each node is represented by its two-dimensional locational code. The binary search based function ACCESSING( ), discussed in chapter three, is used to perform the accessing process.

The accessing algorithm is closely related to the performance of the flight path planning. A sequence of collision checking is performed during the obstacle region extraction and visibility graph construction processes, where binary search accessing is employed with a performance $O(logN_L)$ for the major operations of the collision checking process. Table 6.6 presents the time performance for accessing the nodes along a set of lines randomly generated by Bresenham's algorithm at different layers of a pyramid.

Number of nodes	No. of comparison per retrieve	mean time per retrieve
33253	15.02	301 μsec
16717	14.03	287 μsec
8659	13.08	272 μsec
3217	11.65	258 μsec
961	9.91	240 μsec
253	7.98	202 μsec

Table 6.6. Time performance of the binary search method based retrieval of terrain oct-tree.

It is clear that the number of key value comparisons in a binary search routine increases with the number of nodes in a oct-tree (refer to Figure 3.8). In addition to the conversion of co-ordinates to locational codes, each node accessing operation requires n key value comparisons in the binary search routine and each comparison needs one computation of the projection codes of each search key, where n = $O(logN_L)$.

For example, as shown in Table 6.6, it takes 14 comparisons to retrieve a node from a list of 16717 nodes ($\log_2 16717 = 14.0289$) and 8 comparisons to retrieve a node from a list of 253 nodes ($\log_2 253 = 7.983$). The results show that the average time to retrieve a node from a list is in the range 0.21 ms to 0.31 ms depending on the number of nodes in the tree. The mean time performance was evaluated by timing 256,000 retrieval operations from each terrain oct-tree.

Figure 6.14 depicts the performance of the accessing algorithm with respect to the number of nodes in the search space, the number of key value comparisons during a node retrieval process and the number of key value comparisons with respect to the time cost of a rerieval process. The number of nodes in the terrain oct-tree determines the accessing performance. The plots in Figure 6.14 match the *O(logn)* property of a binary search routine.

Another operation on terrain oct-trees is the use of locational codes as indices in neighbour finding. Neighbour finding is performed by first computing the locational code of a neighbour node in the projection plan, the oct-tree is then searched for the node (or a larger node) containing the neighbour. The locational code of an adjacent node is obtained by applying equations 3-14 and 3-15 to a node. The computations on locational codes are performed by means of bit manipulations.

6.3.3 Waypoints Location

As mentioned in section 4.4, the number of obstacle nodes varies with the sum of the perimeters of the obstacles, measured in pixel units. When the expansion process is truncated by approximation to level *l* (pixel level l=0), the number of obstacle nodes is reduced by a factor of 2^l, thus reducing the expansion time. The neighbour finding technique is incorporated in the obstacle area expansion and waypoint location algorithms when the adjacent nodes of '*seeds*' are checked recursively against the danger nodes list to determine the boundary nodes.

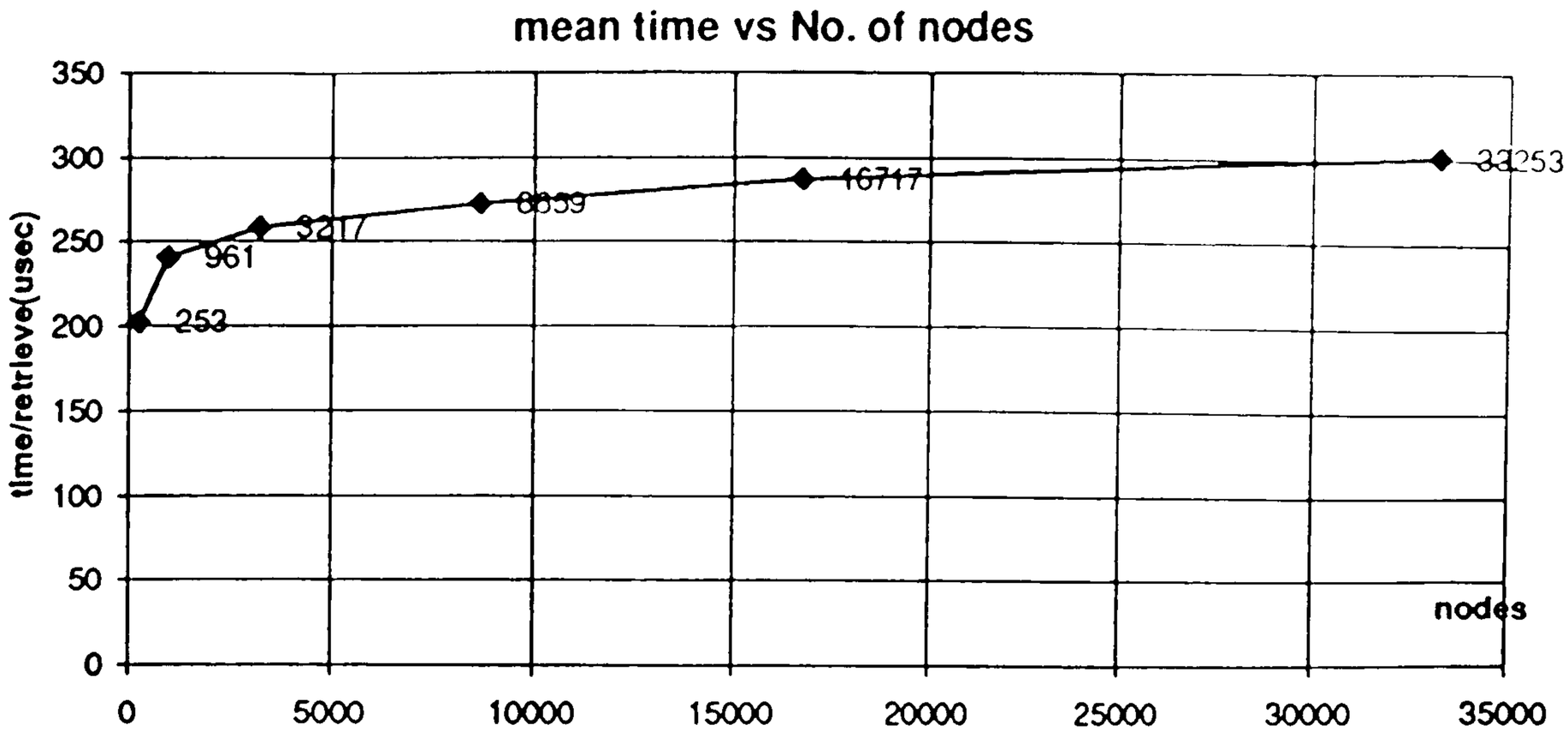


Figure 6.14a Time performance to retrieve a node from a list of oct-tree nodes.

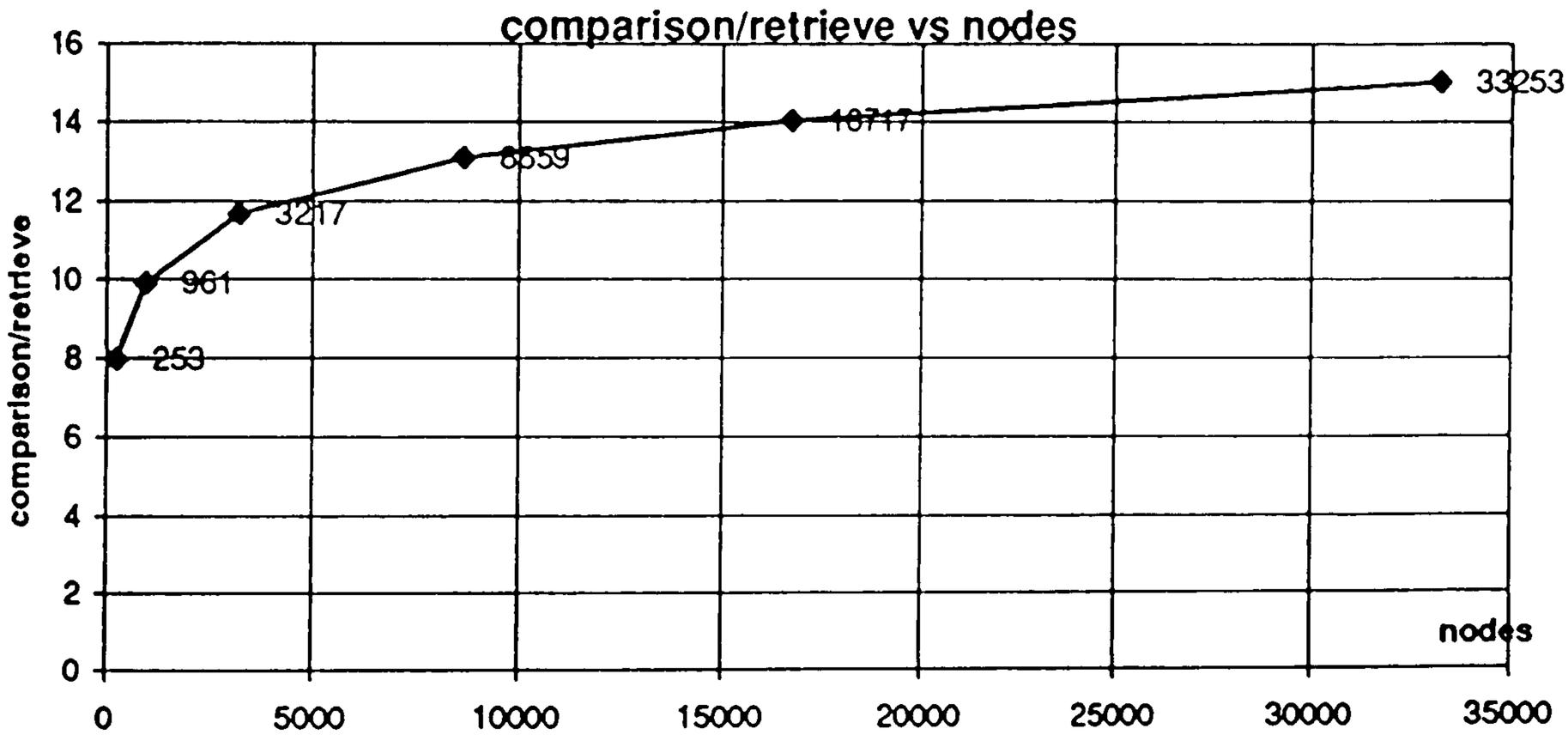


Figure 6.14b The number of executing the key value comparisons during a node retrieval process

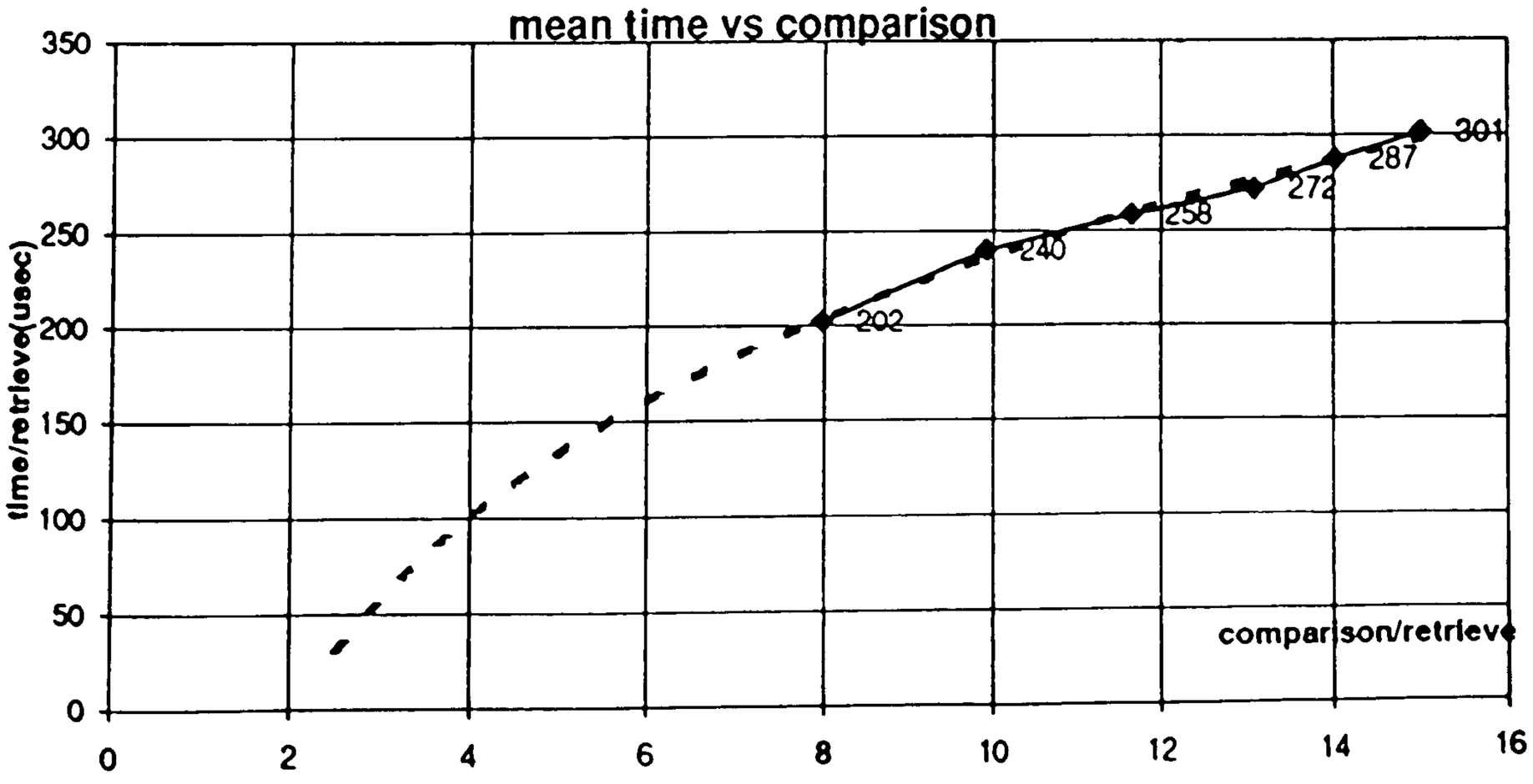


Figure 6.14c Time to retrieve a node vs No. of key comparison.

Figure 6.15 depicts an example of the obstacles regions derived by this expansion process at both the finest and a predefined level of boundary nodes. The derived obstacle regions are highlighted together with the waypoints at each layer of a terrain pyramid; the number of waypoints and the time cost to derive the waypoints are also given in Figure 6.15.

Table 6.7(a,b) presents the time cost with respect to the number of danger nodes at different layers of a pyramid during the waypoint location process. As described in chapter 5, the number of nodes at a layer of a pyramid is about a quarter of the number of nodes in its predecessor layer (using a 2 x 2 approximation window); expanding the obstacles at a resolution *l* layer above the pixel resolution (where pixel resolution is defined as layer 5 in Figure 6.15a) reduces both the number of danger nodes and obstacle nodes, substantially reducing the time cost of the expansion process.

6.3.4 Visibility Graph Construction

The major task in the transformation stage is to use the collision checking technique to construct a visibility graph of the navigation space. The number of path segments to be checked depends on the number of waypoints. The collision check determines if there is an intersection between a path segment and two waypoints. The performance of the binary search method of collision checking algorithm depends on the number of nodes along the path segment and the size of the danger nodes list used in the binary search. The time cost of the entire collision check is $O(N_{lp}*logN_{dn})$, where N_{lp} is the number of point elements along a direct path, measured in resolution units and N_{dn} denotes the number of danger nodes described in section 4.4.2.

In the worst case of collision checking, each node along a path segment is searched against the danger node list. On the other hand, the best case takes only one binary search to verify the intersection. The time of the collision check process is proportional to the length of a path segment. Figure 6.16 gives examples of performing the path planning at different resolution layers of a terrain pyramid; as shown in these

examples, the number of danger nodes, the number of node elements along a line segment and the number of waypoints obtained decrease as the resolution level is reduced. The results show that waypoints are located in an upper layer of a pyramid much faster than in a lower layer. Since the number of path segments increases with the number of waypoints, the resultant visibility graph will generate different flight paths.

Table 6.7 illustrates the timing of the visibility graph construction at different layers of two test pyramids. Accordingly, Figures 17 and 18 depict the time performance of each stage as well as the overall cost of the planning process with respect to the number of waypoints in the navigation space. The results show that the time cost for visibility graph construction, based on 20 waypoints (and less), is within five seconds which complies with the real-time constraints described in chapter 5 (Table 5.1). Moreover, the real-time constraints can be varied by flight conditions, for example, increasing the time constraint to 10 seconds in which the waypoints are less than 40 (refer to Table 5.1) and the aircraft speed is 200 m/sec.

DTED source file : 512 x 512 grid points of Peak District area.
Oct-tree file : TABLE 6.7a PEAKS710.OCT; TABLE 6.7b PEAKS810.OCT.
Flight Altitude : 600 metres.
Scaling Factor : 100 metres.
No. of Nodes : The number of nodes in the terrain oct-tree.
Danger Nodes : The number of nodes above predefined flight altitude.
W_{points} : The number of waypoints.
$P_{segment}$: The number of valid path segments.
Start and Goal : (255,0), (0,255).
T_{WP} (sec) : The time to locate the waypoints.
T_{VG} (sec) : The time to construct the visibility graph.
T_{SP} (sec) : The time to search a path in the visibility garph.
Path Distance : Distance from the start to the goal points.
Total Cost (sec) : Total time to find a path.

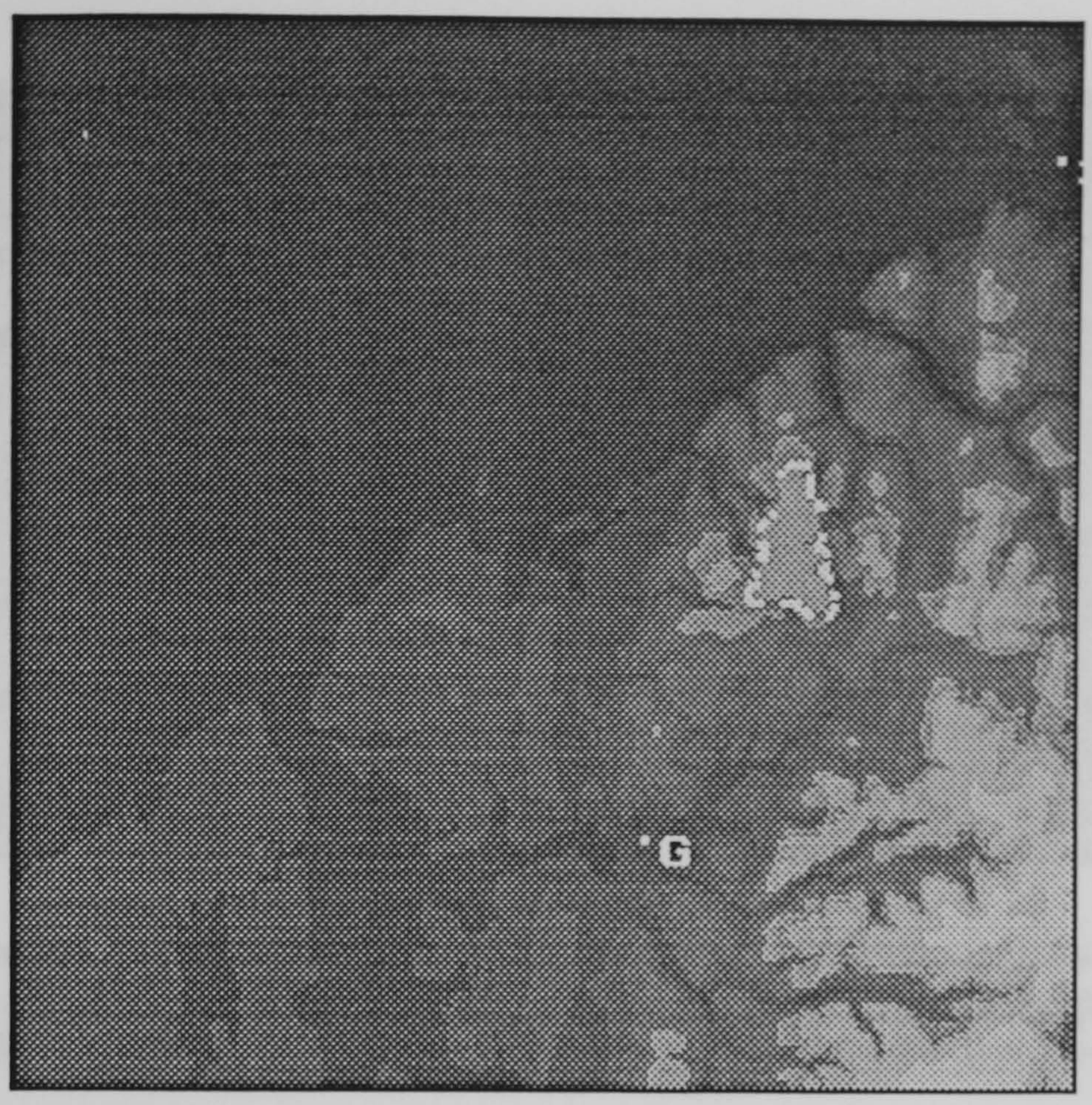
Layer	No. of Nodes	Danger nodes	W_{points}	$P_{segment}$	T_{WP} (sec)	T_{VG} (sec)	T_{SP} (sec)	Total (sec)	Dist (m)
5	6448	110	48	480	0.367	11.392	0.367	12.126	36800
4	2938	72	22	150	0.220	2.527	0.275	3.022	36000
3	940	37	11	50	0.110	0.495	0.110	0.715	36200
2	253	16	7	26	0.055	0.165	0.110	0.330	37300
1	64	8	6	20	0.055	0.110	0.055	0.220	42700

(9a)

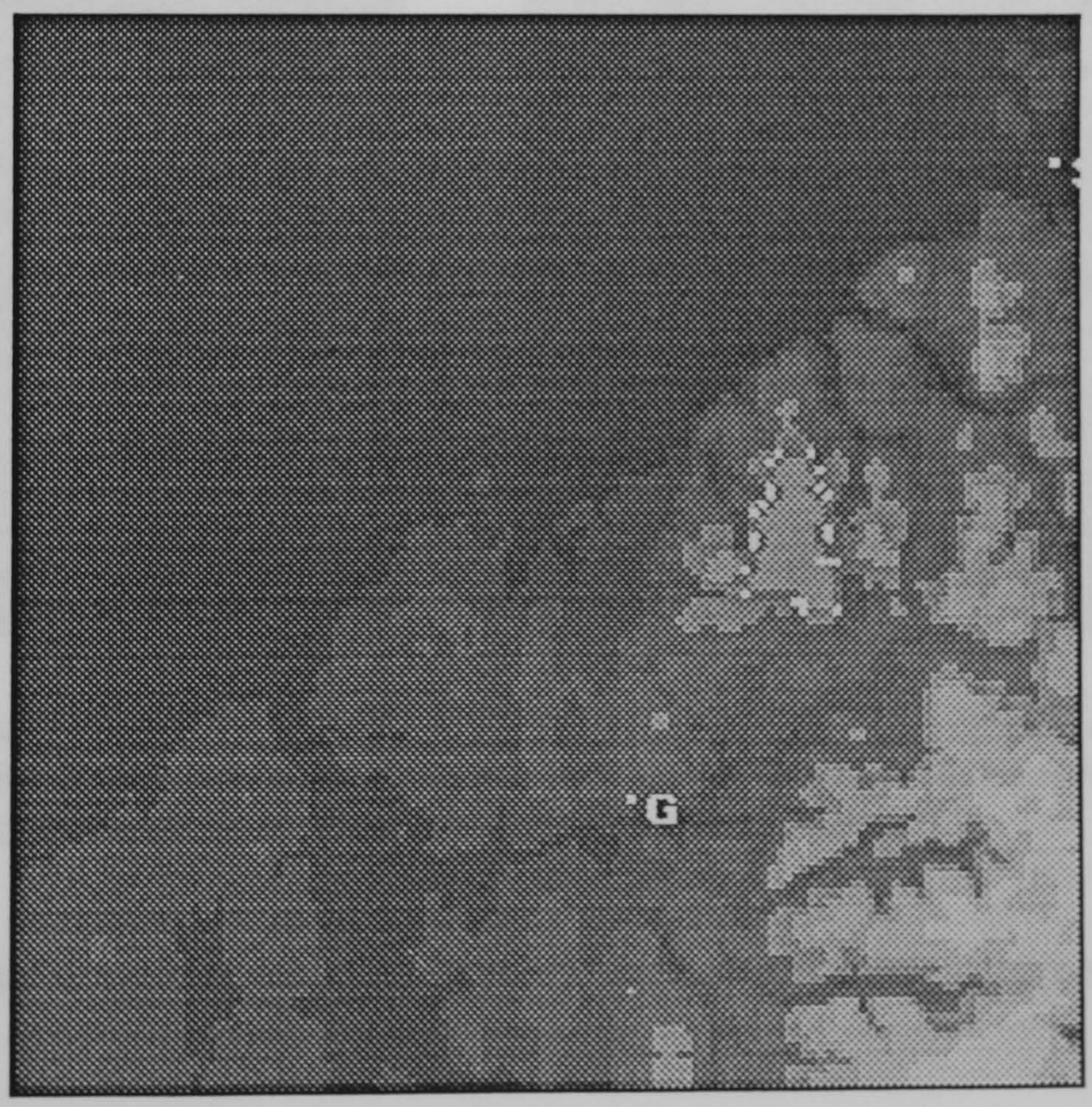
Layer	No. of Nodes	Danger nodes	W_{points}	$P_{segment}$	T_{WP} (sec)	T_{VG} (sec)	T_{SP} (sec)	Tatal (sec)	Dist (m)
5	16717	250	65	844	0.769	35.973	1.392	38.134	36800
4	8659	182	40	348	0.659	11.484	0.220	12.363	36100
3	3217	78	20	130	0.220	2.363	0.165	2.748	36200
2	961	37	11	50	0.110	0.549	0.110	0.769	36500
1	253	16	7	26	0.055	0.110	0.110	0.275	37200

(9b)

TABLE $6.7_{a,b.}$ The experimental results of applying the flight path planning algorithm on 2^9x2^9 DTM file encoded terrain oct-trees pyramid in static environment. The DTM files are reduced to 2^7x2^7 and 2^8x2^8 by a 4x4 and 2x2 approximation window respectively before they are encoded as terrain oct-trees.

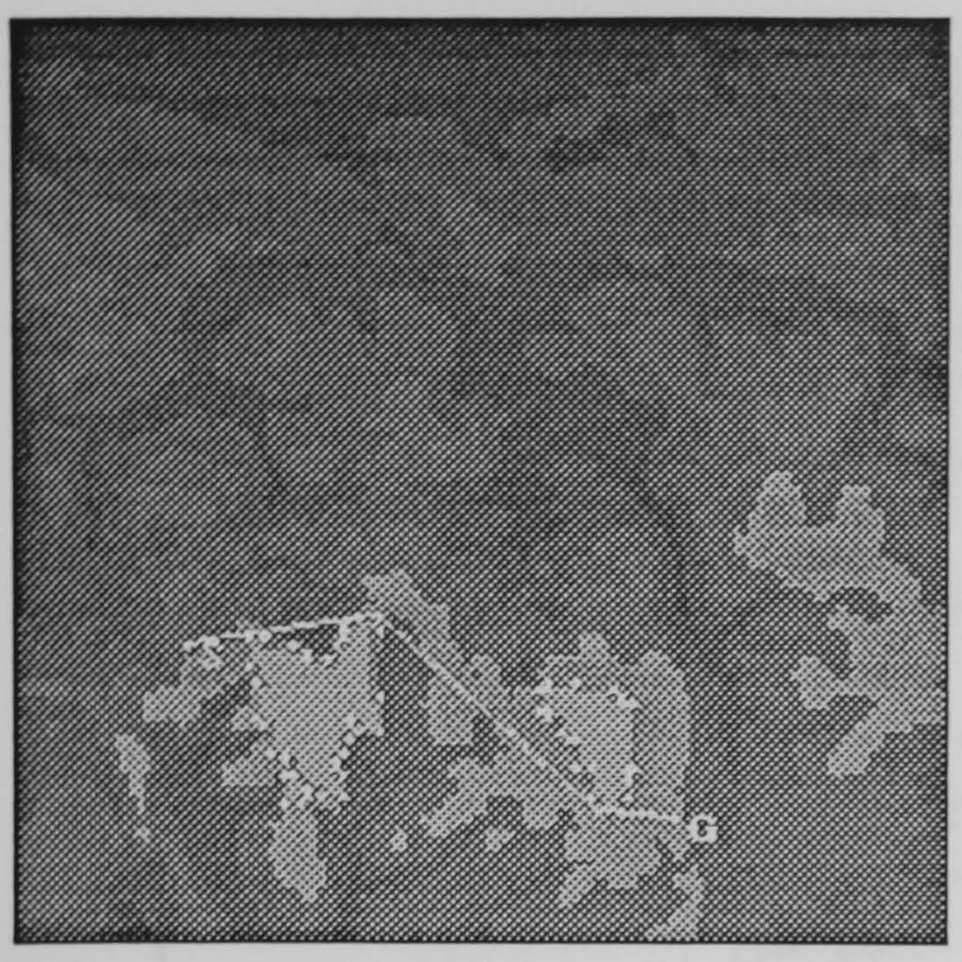


(a) 49 waypoints around an obstacle region obtained by expansion of the finest resolution level . Expansion time 0.989 sec.

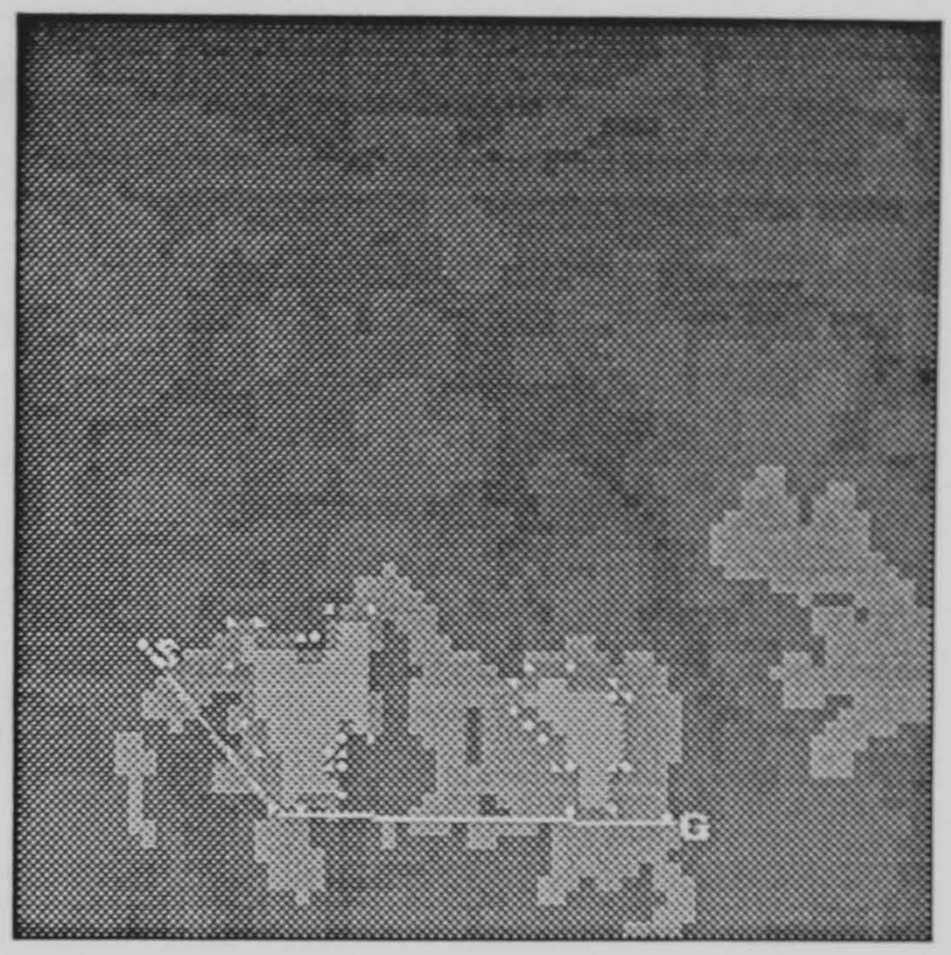


(b) 25 waypoints around obstacle region obtained by one level truncated expansion. expansion time 0.604 sec.

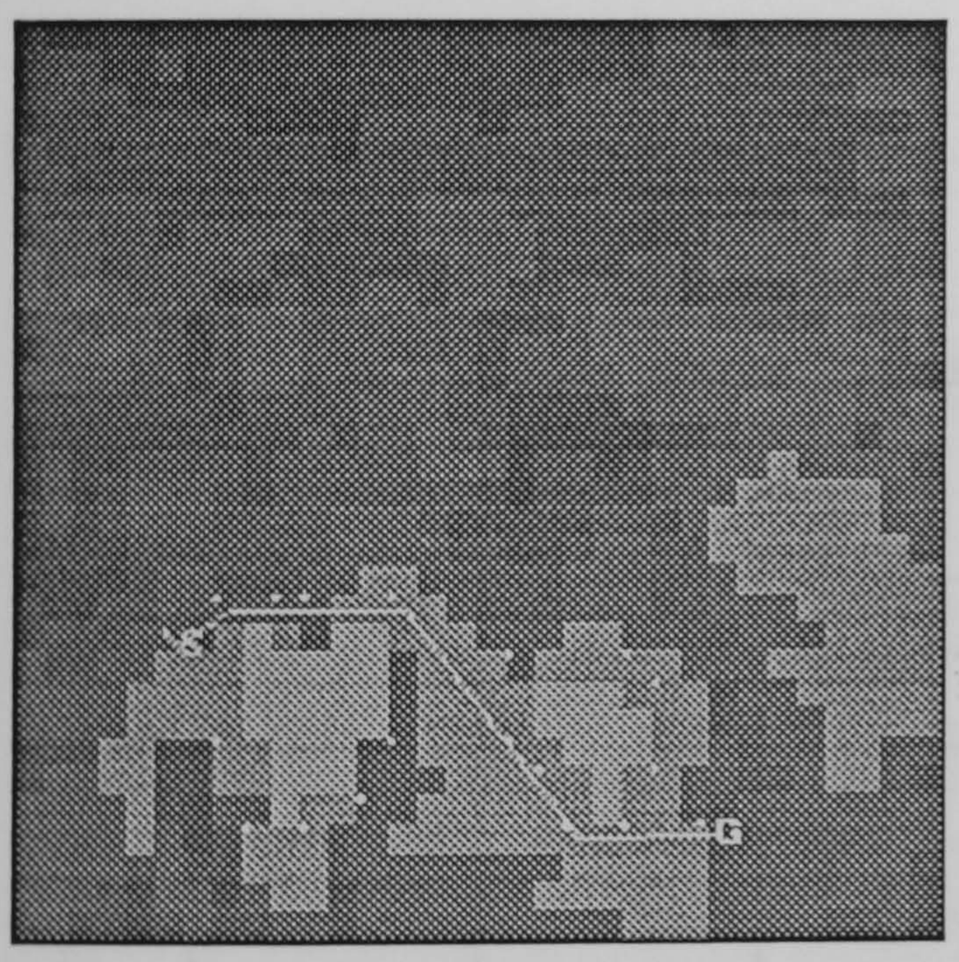
Figure 6.15 Example of waypoint location during expansion of the obstacle regions.



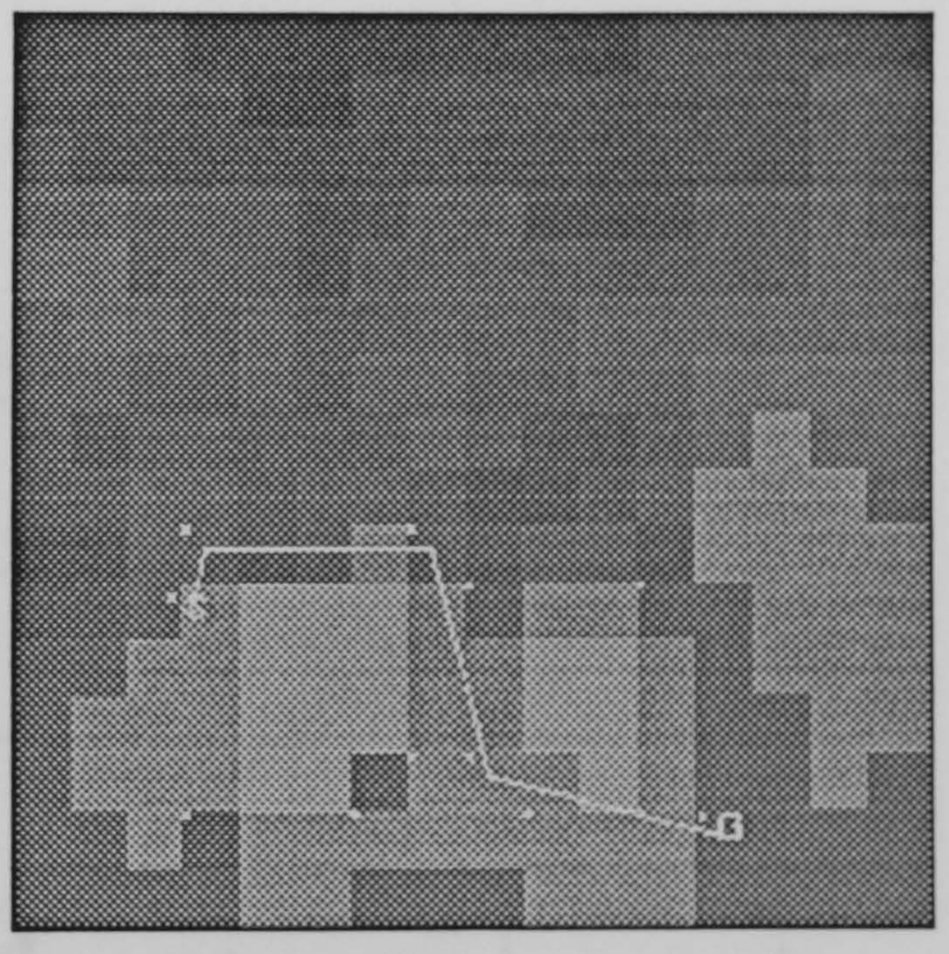
(a) 65 waypoints, 844 path segments at layer 5, 16717 nodes in the tree, 250 danger nodes, 1.099 sec to locate the waypoints.



(b) 35 waypoints, 372 path segments at layer 4, 8659 nodes in the tree, 182 danger nodes, 0.445 sec to locate the waypoints.



(c) 19 waypoints, 124 path segments at layer 3, 3217 nodes in the tree, 78 danger nodes, 0.165 sec to locate the waypoints.



(d) 12 waypoints, 64 path segments at layer 2. 961 nodes, 37 danger nodes, 0.055 sec to locate the waypoints.

Figure 6.16 The obstacle regions, waypoints and flight path at different layers of a 100 metres scaling factor encoded oct-tree terrain pyramid.

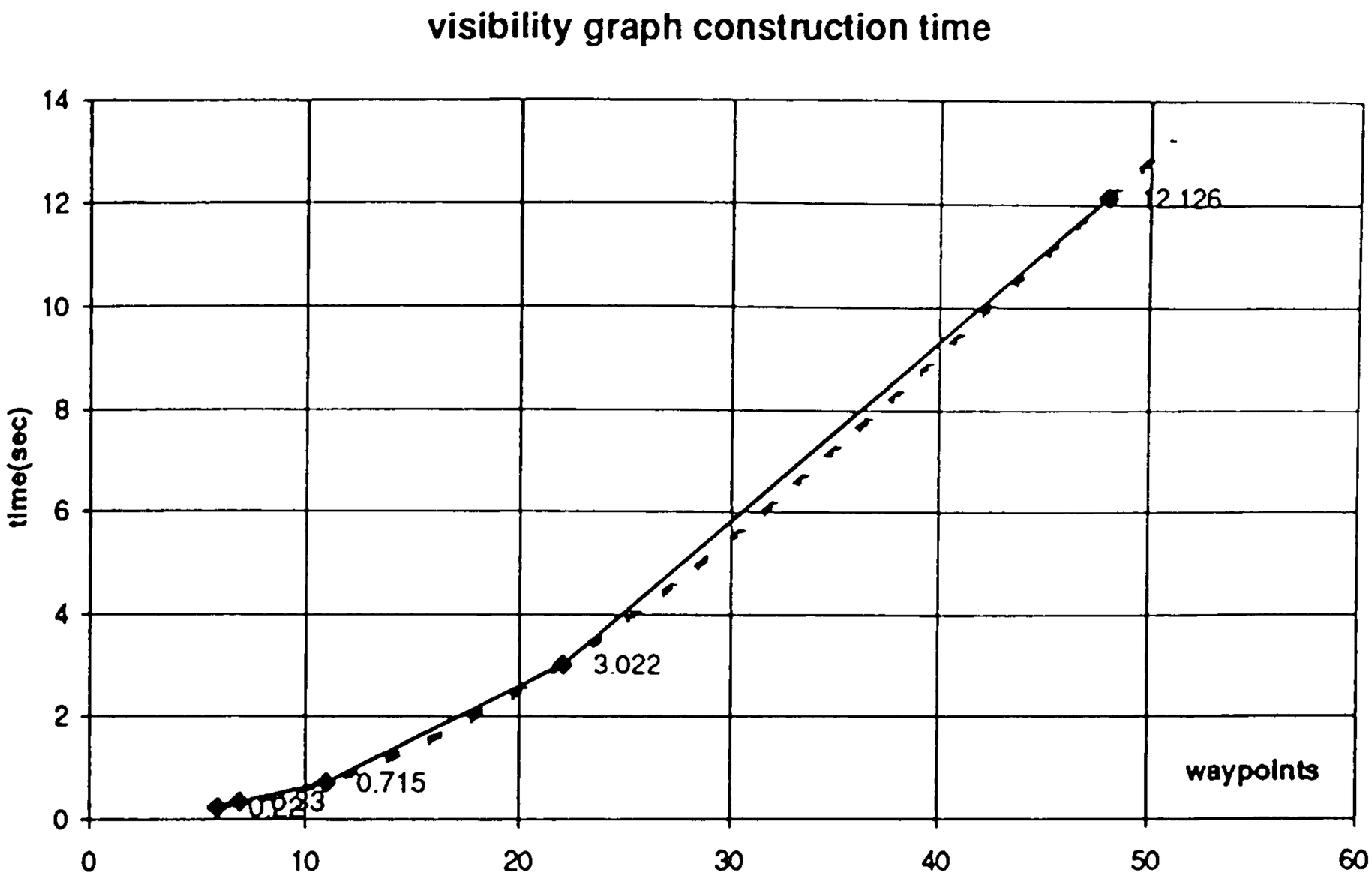


Figure 6.17a Visibility graph construction time vs number of waypoints.

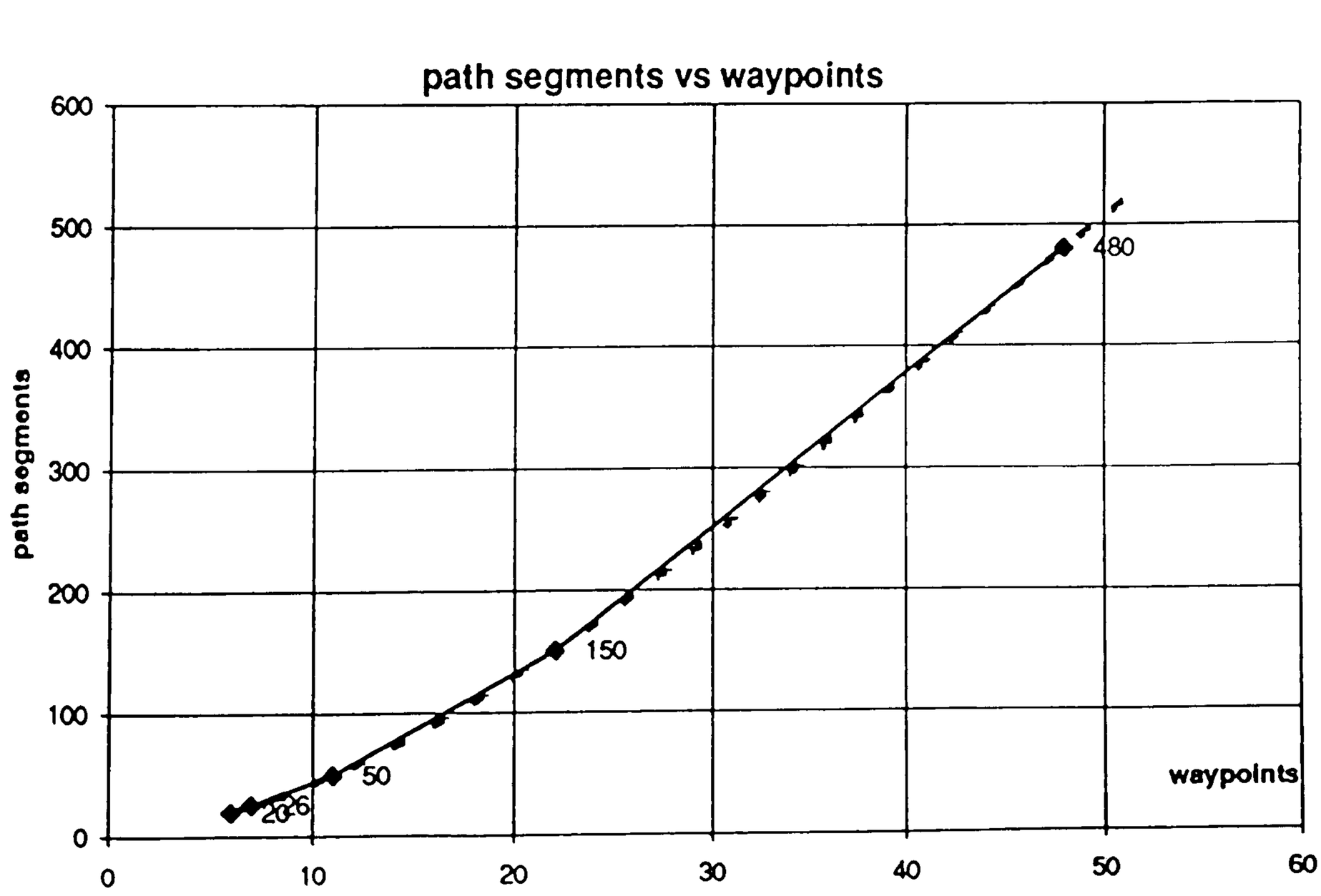


Figure 6.17b Path segments vs waypoints.

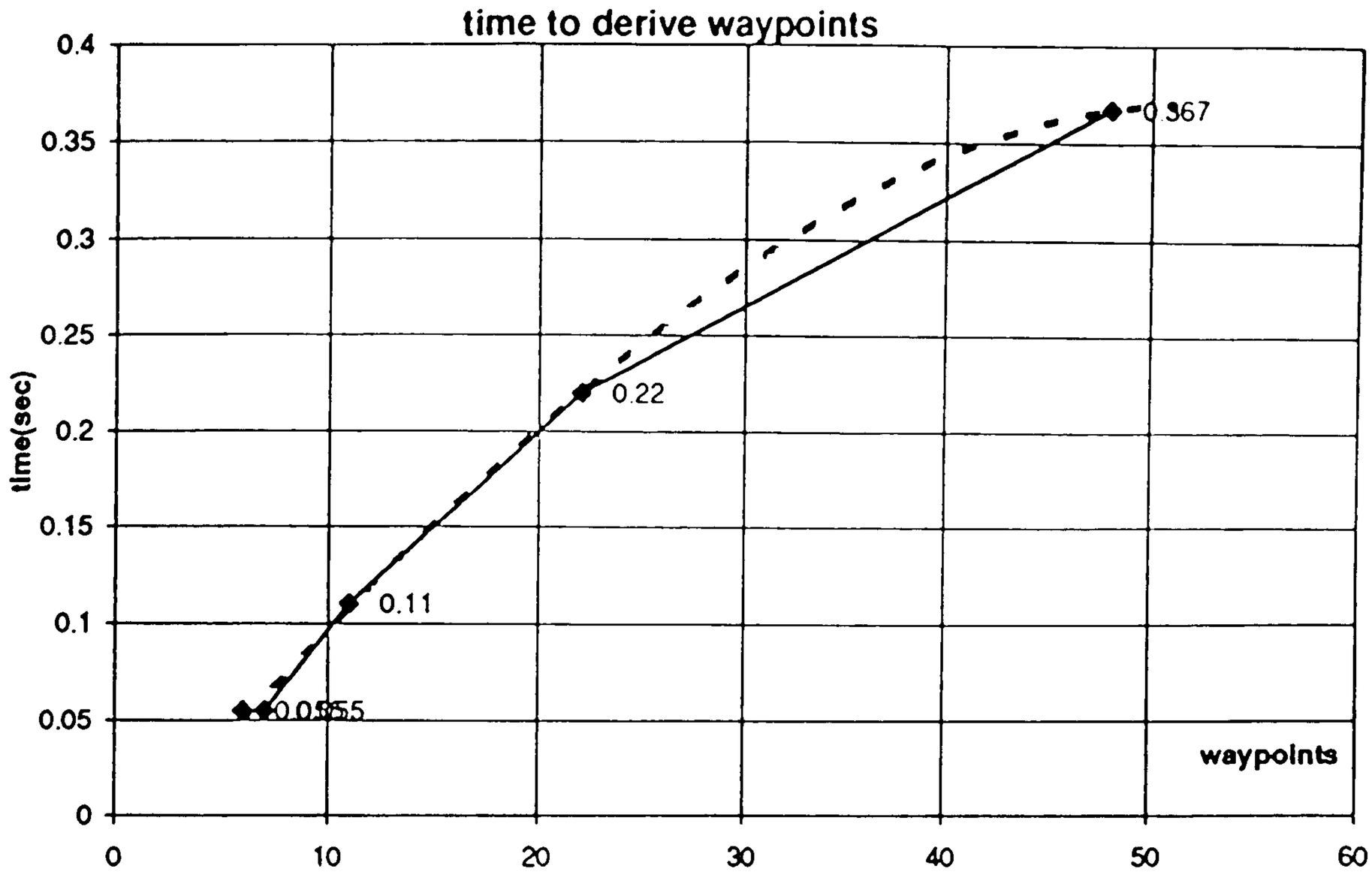


Figure 6.17c Derive time vs number of waypoints.

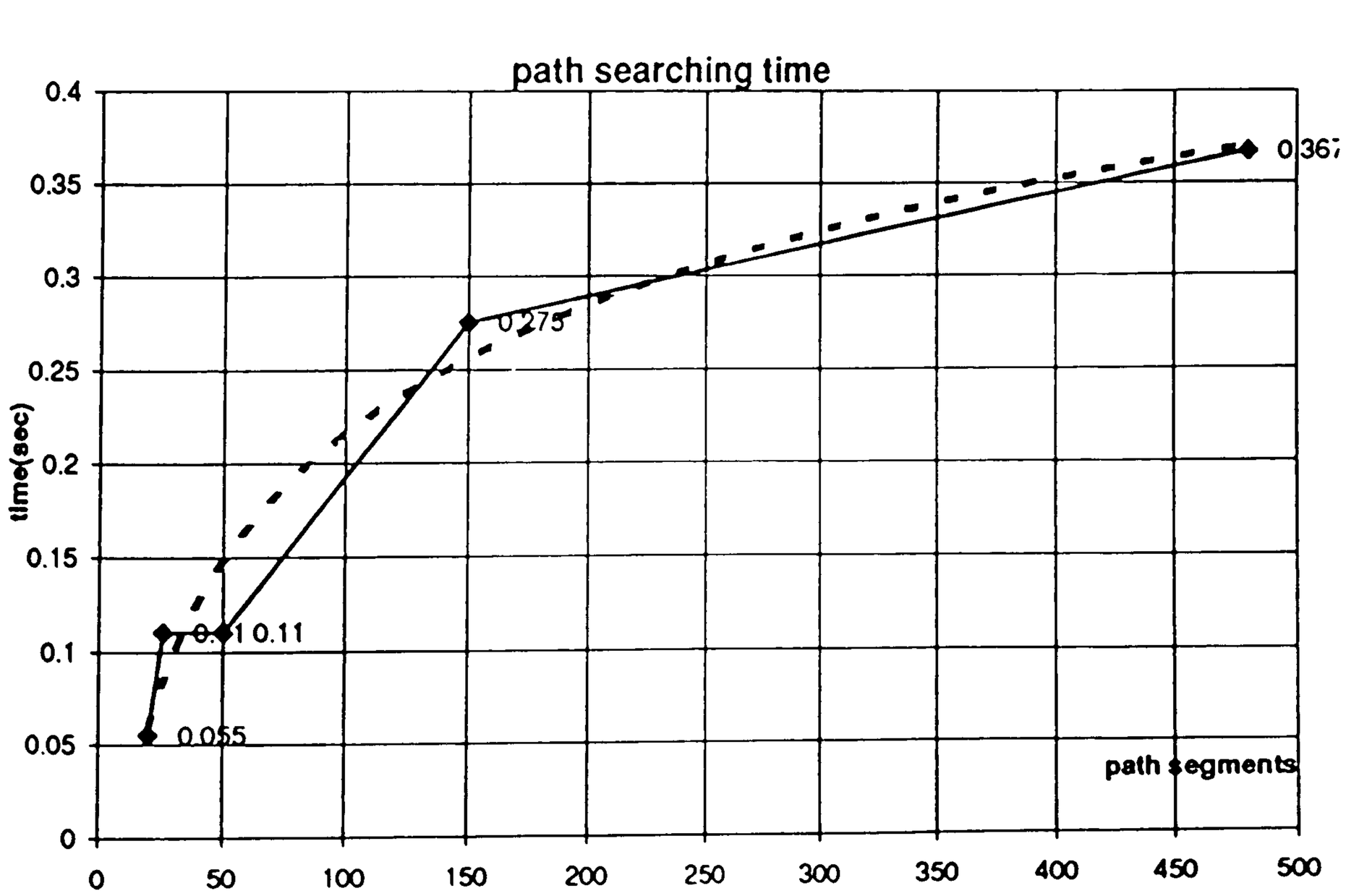


Figure 6.17d Time to obtain a path from the visibility graph.

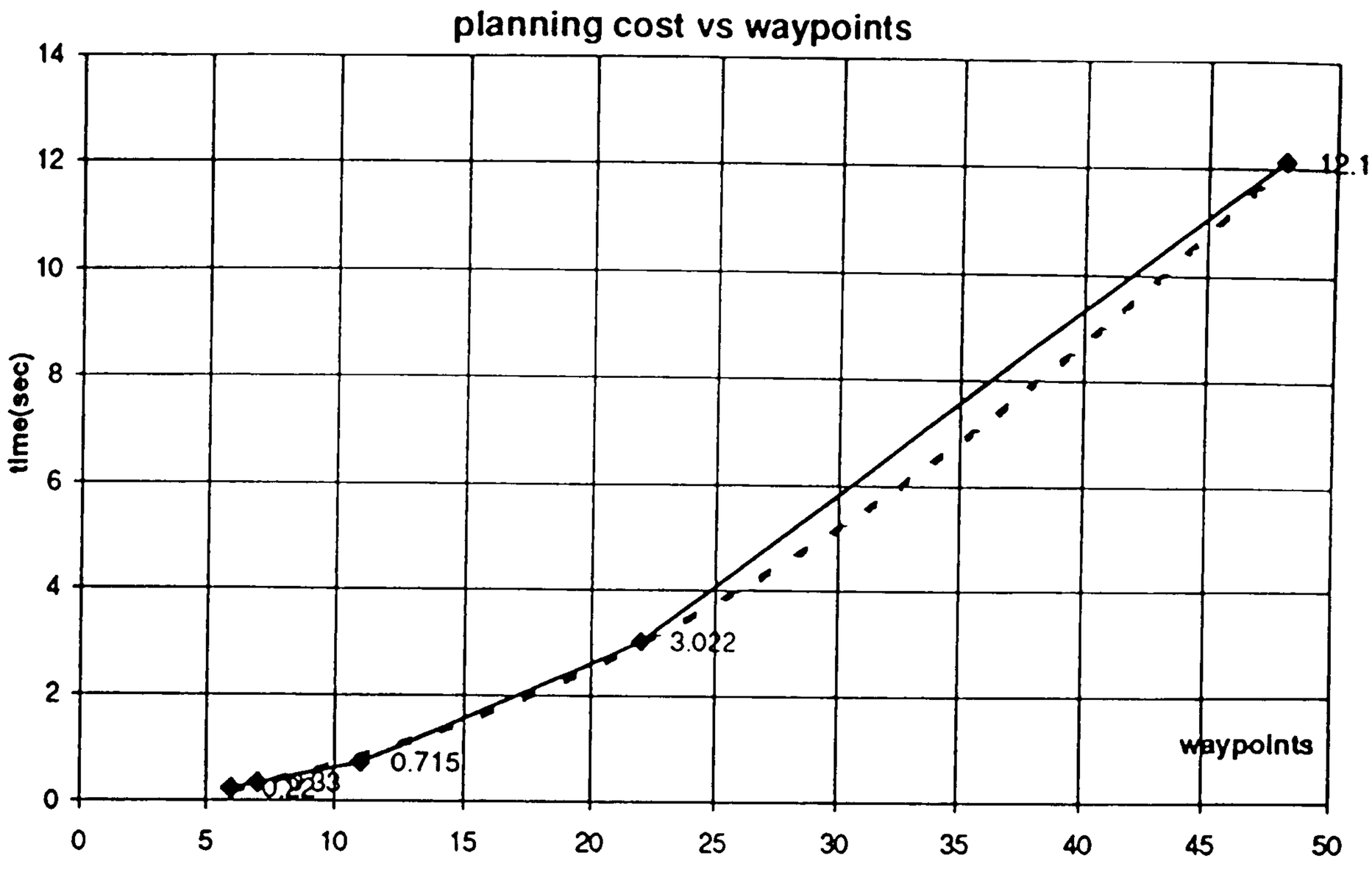


Figure 6.17e Total time cost of path planning vs number of waypoints.

Figure 6.17 Time performance analyses refer to Table 6.7a.

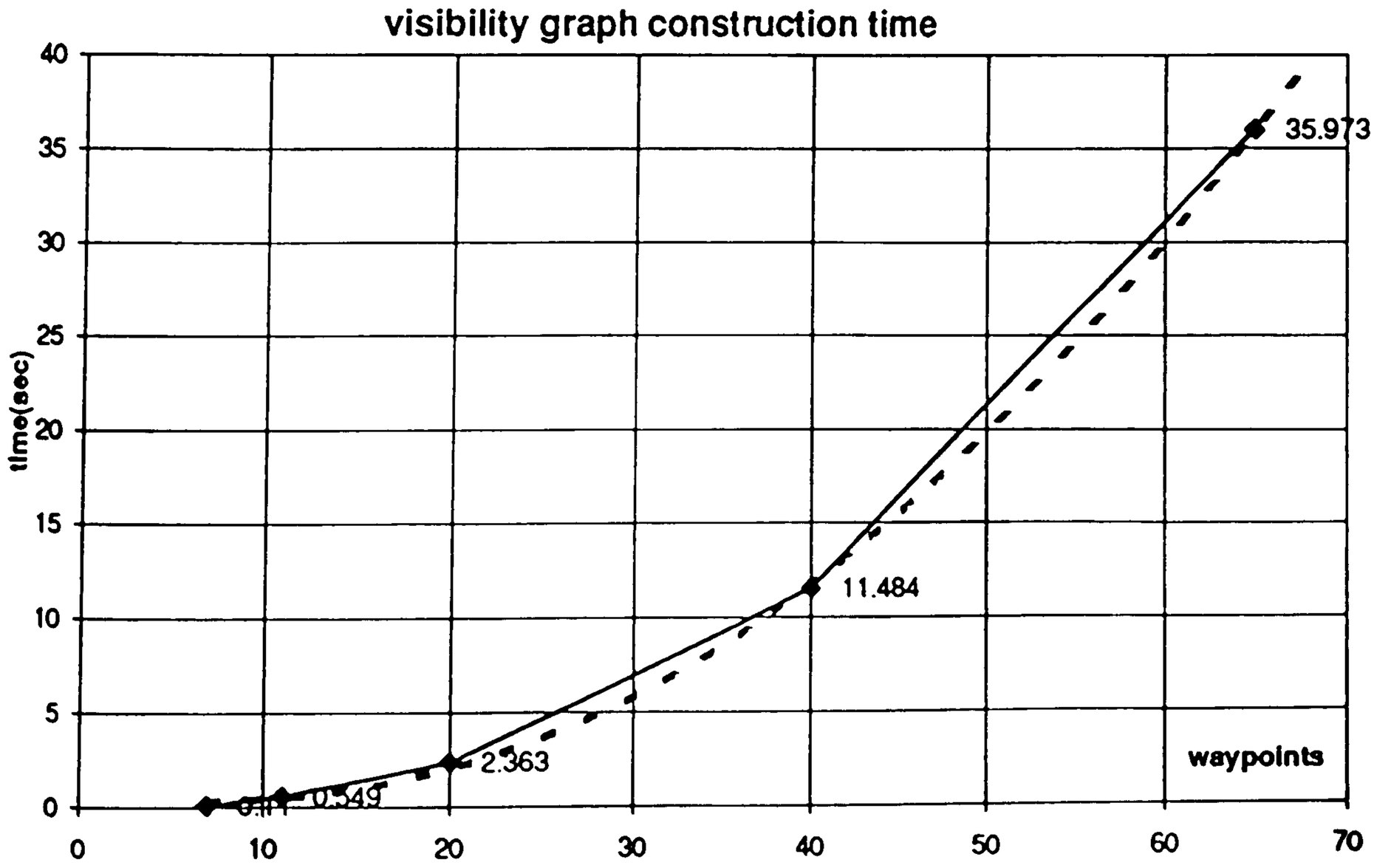


Figure 6.18a Visibility graph construction time vs number of waypoints.

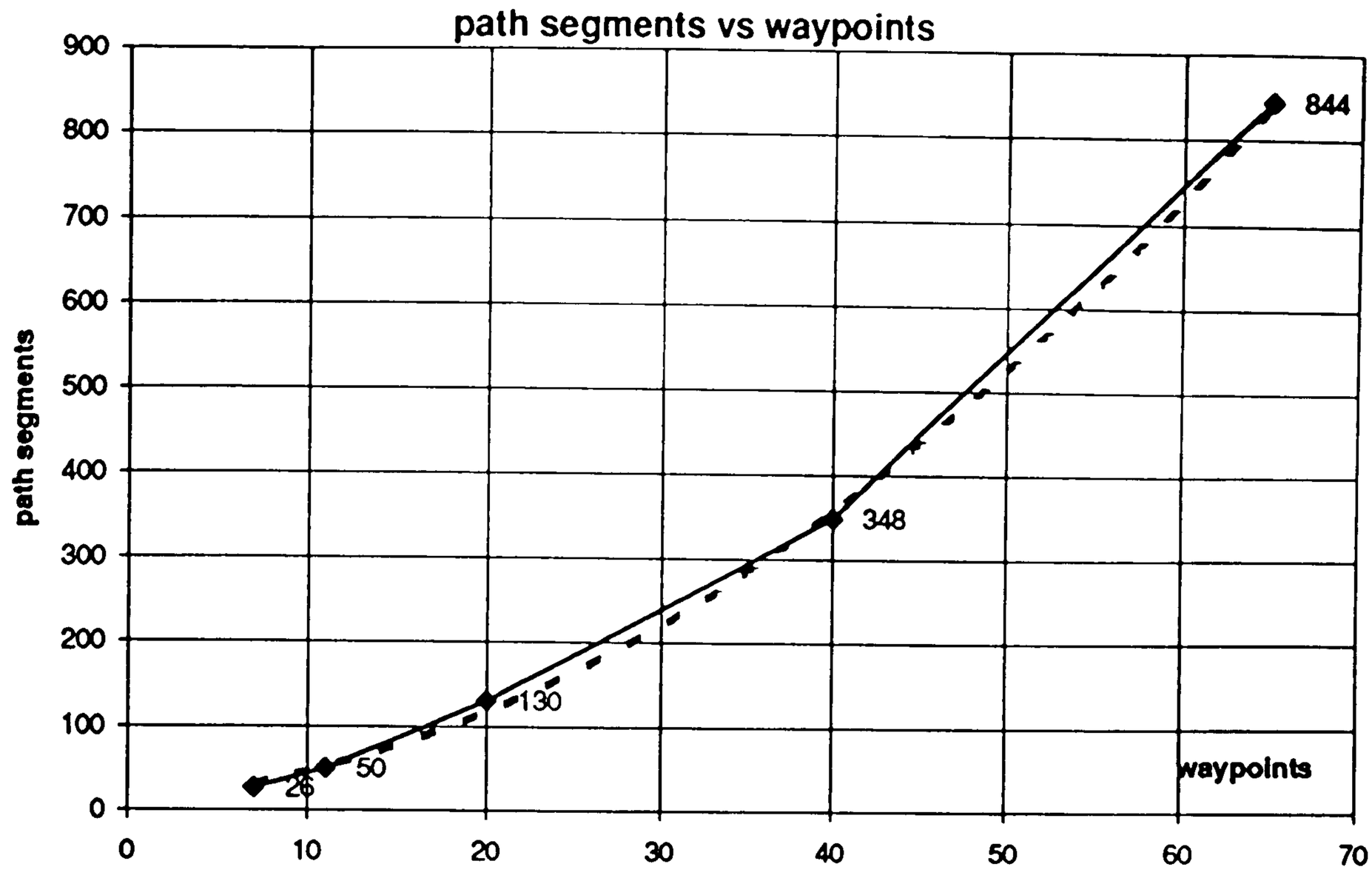


Figure 6.18b Path segments vs waypoints.

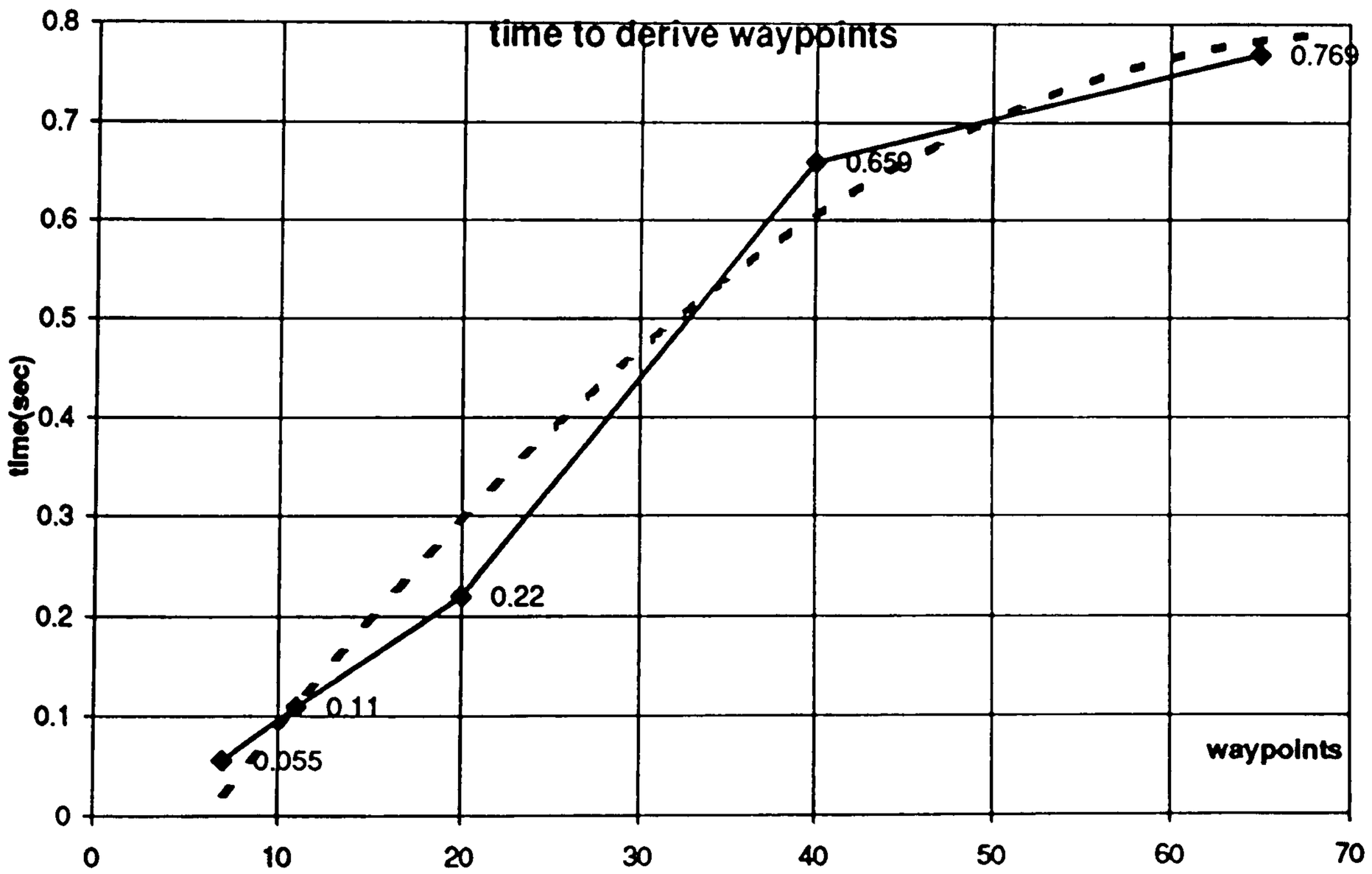


Figure 6.18c Derive time vs number of waypoints.

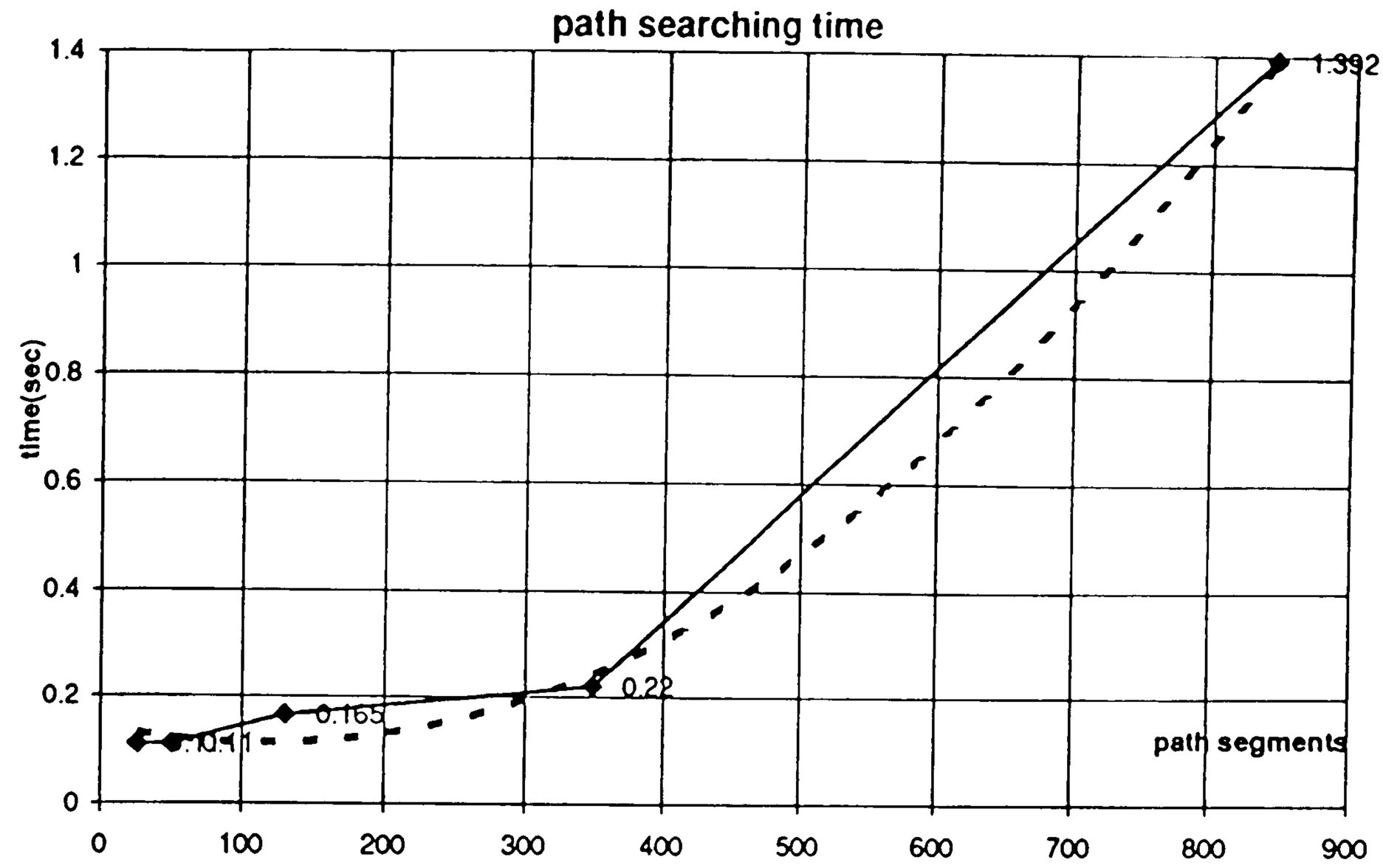


Figure 6.18d Time to obtain a path from the visibility graph.

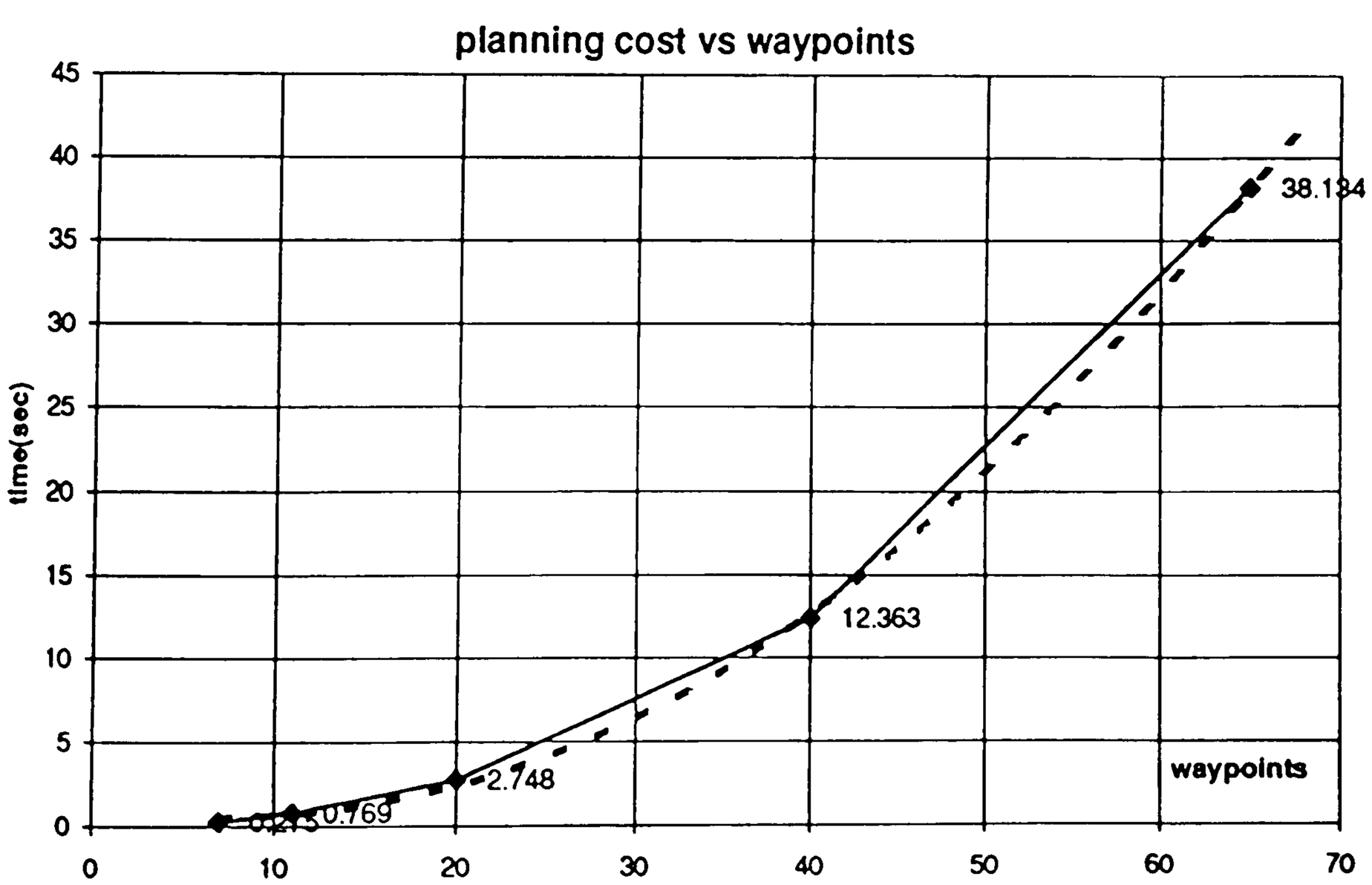


Figure 6.18e Total time cost of path planning vs number of waypoints.

Figure 6.18 Time performance analyses refer to Table 6.7b.

6.3.5. Path Searching

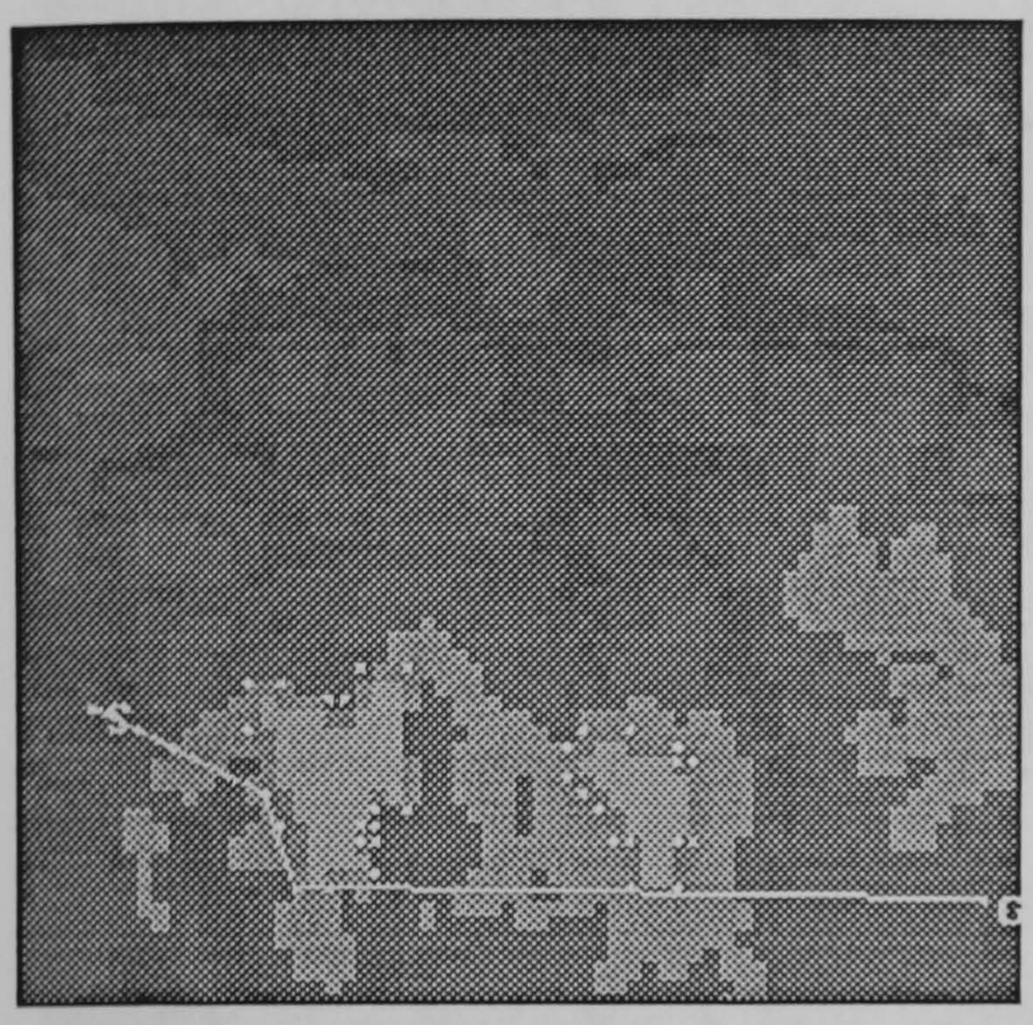
In the path searching stage, the visibility graph is represented as a list of flight path segments. Each record in the list contains the information of a valid arc (path segment) in the graph, which includes the W_{from} and W_{to} locational codes, the distance of a path segment and a backtracking flag, as shown in Figure 4.11.

Various search methods have been adopted to demonstrate the generality of the data structure which is used to represent the visibility graph and the generality of locational code data format which is used to depict the co-ordinates of a spatial point. Figure 6.19 gives examples of multiple solutions using a depth-first search and path removal method at layer three in the terrain oct-tree navigation space. The optimal path is obtained by keeping the one with shortest flight distance (Figure 6.19c) among the multiple solutions as described in Section 5.5.

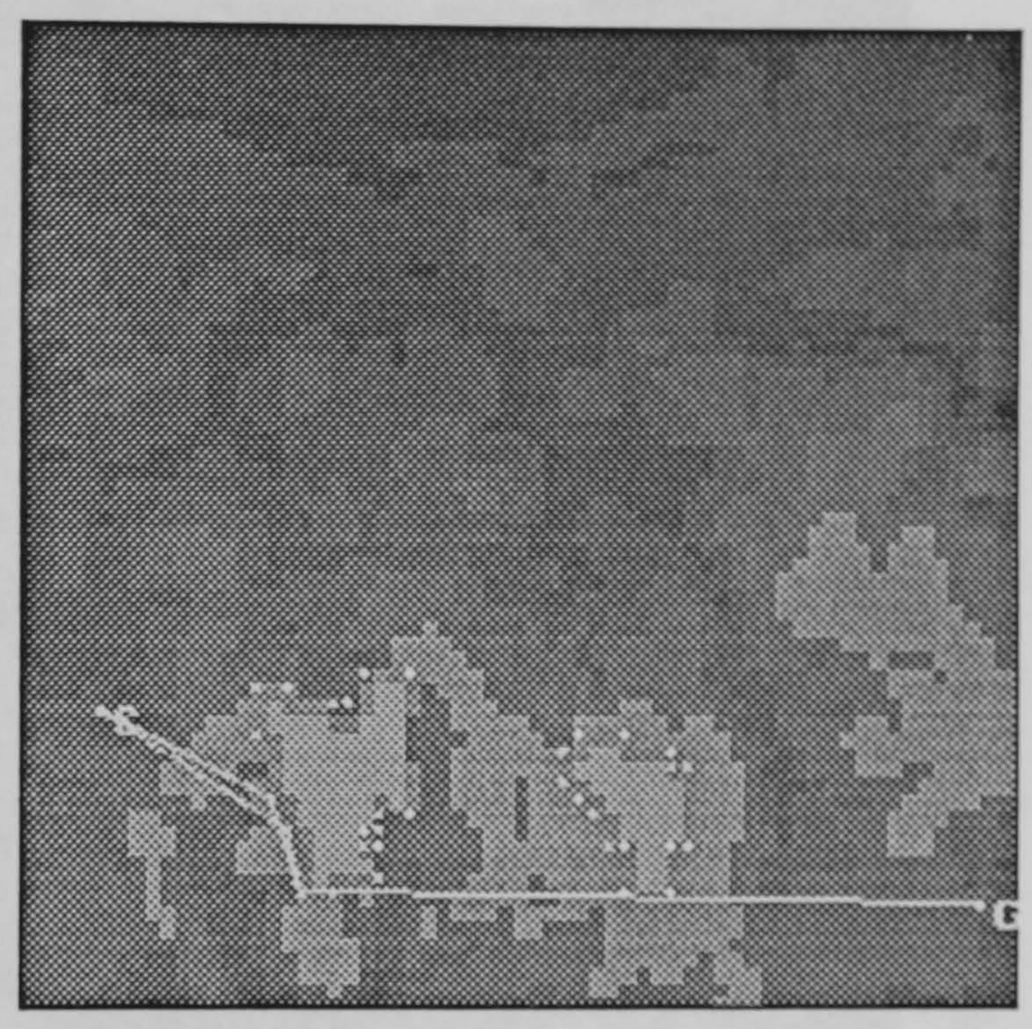
Figure 6.20 depicts the results based on a variant of the A* search method which uses path distance as a cost figure together with a least cost heuristic to find the set of optimal solutions. In this thesis, the cost function is simplified to flight distance only. Figures 6.21 and 6.22 depict the results of performing path planning at different layers of a terrain pyramid in a 'static' navigation environment. The table of each figure illustrates the performance of the flight planning algorithms for a wide range of different baseline elevations and scaling factors in encoded terrain oct-trees.

6.4 Results from Real-Time Dynamic Flight Path Planning

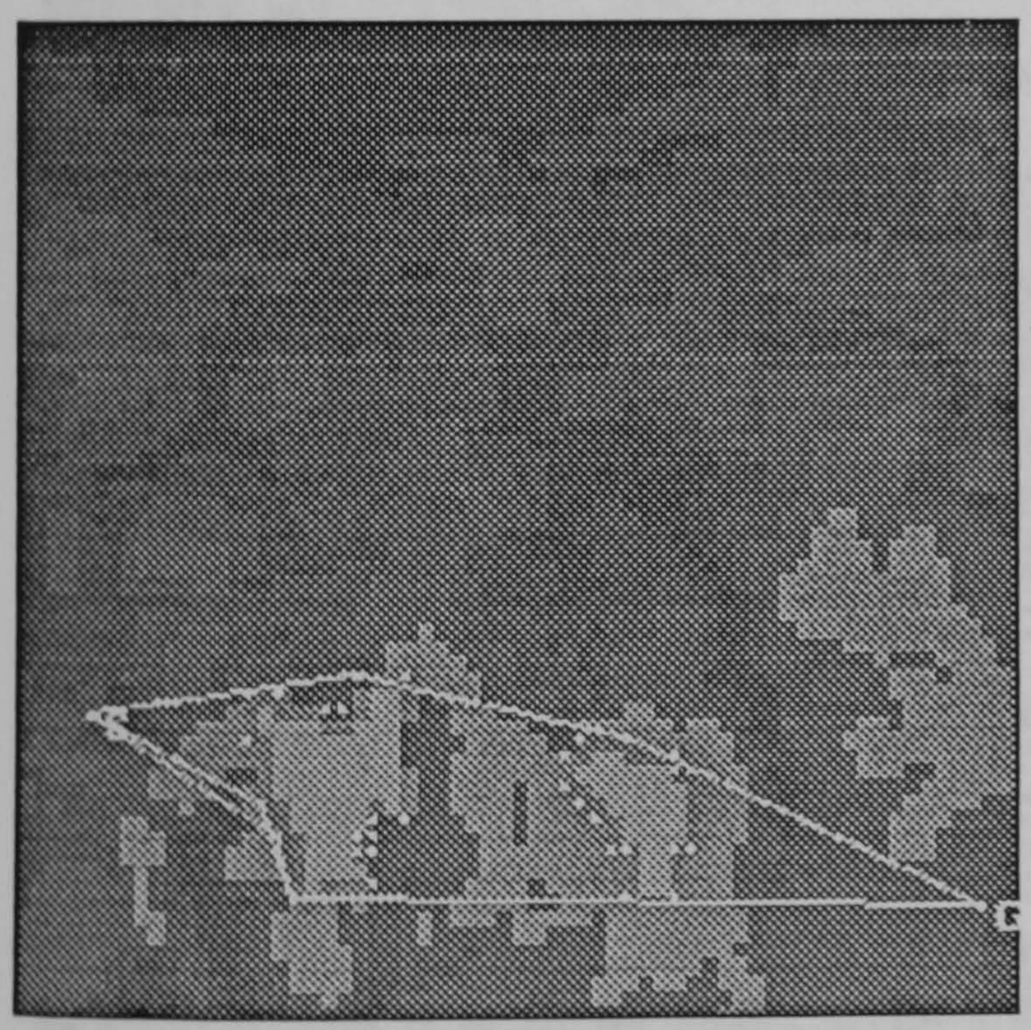
The observations obtained from executing the flight path planning algorithm in a static environment have been integrated into a PC-based simulation of real-time dynamic path planning. The approach to real-time simulation has been described in chapter 5. The experimental results are as follows:



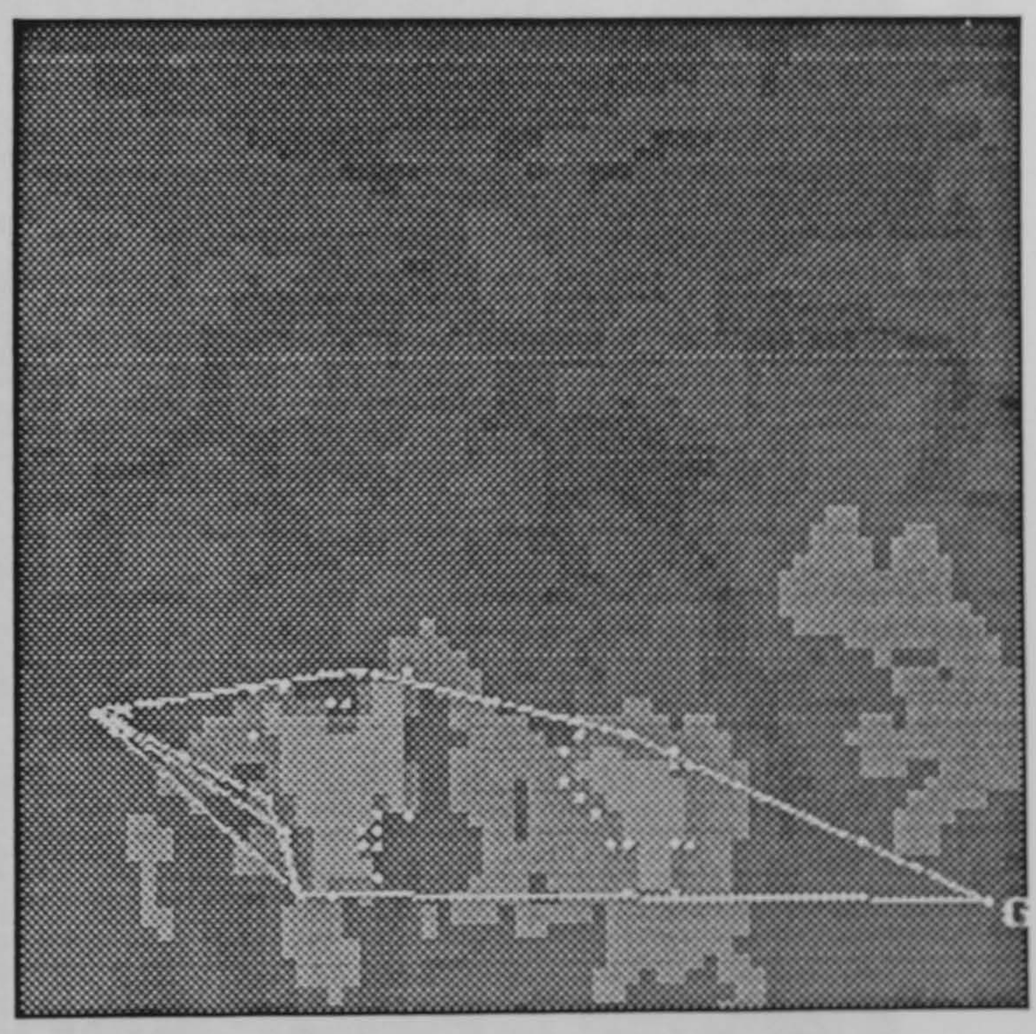
(a) Path one, from (19,173) -> (64,196) -> (72,220) -> (245,224), distance 24.8 km.



(b) Path two, from (19,173) -> (68,204) -> (72,220) -> (245,224), distance 24.7 km.



(c) Path three, from (19,173) -> (88,164) -> (156,180) -> (245,224), distance 23.7 km.



(d) Path four, from (19,173) -> (72,220) -> (245,224) , distance 24.3 km.

Figure 6.19 Multiple path solutions using depth first and least cost heuristic. Layer 3 terrain oct-tree with 8659 nodes, scaling factor 100 metres, minimum flight altitude 600 metres, 182 danger nodes. The shortest path is path 3 in (c).

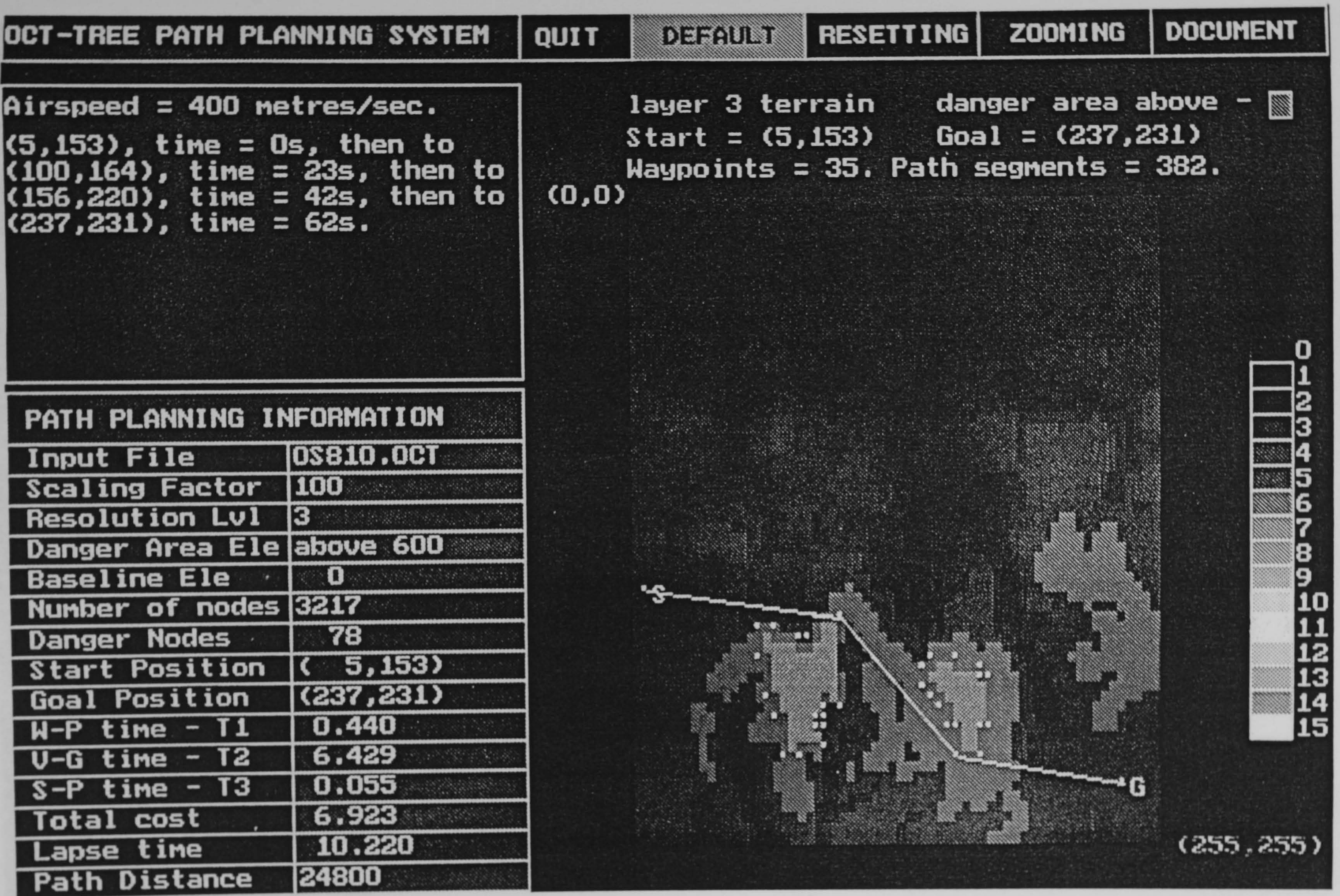


Figure 20a Path searching using A* algorithm.

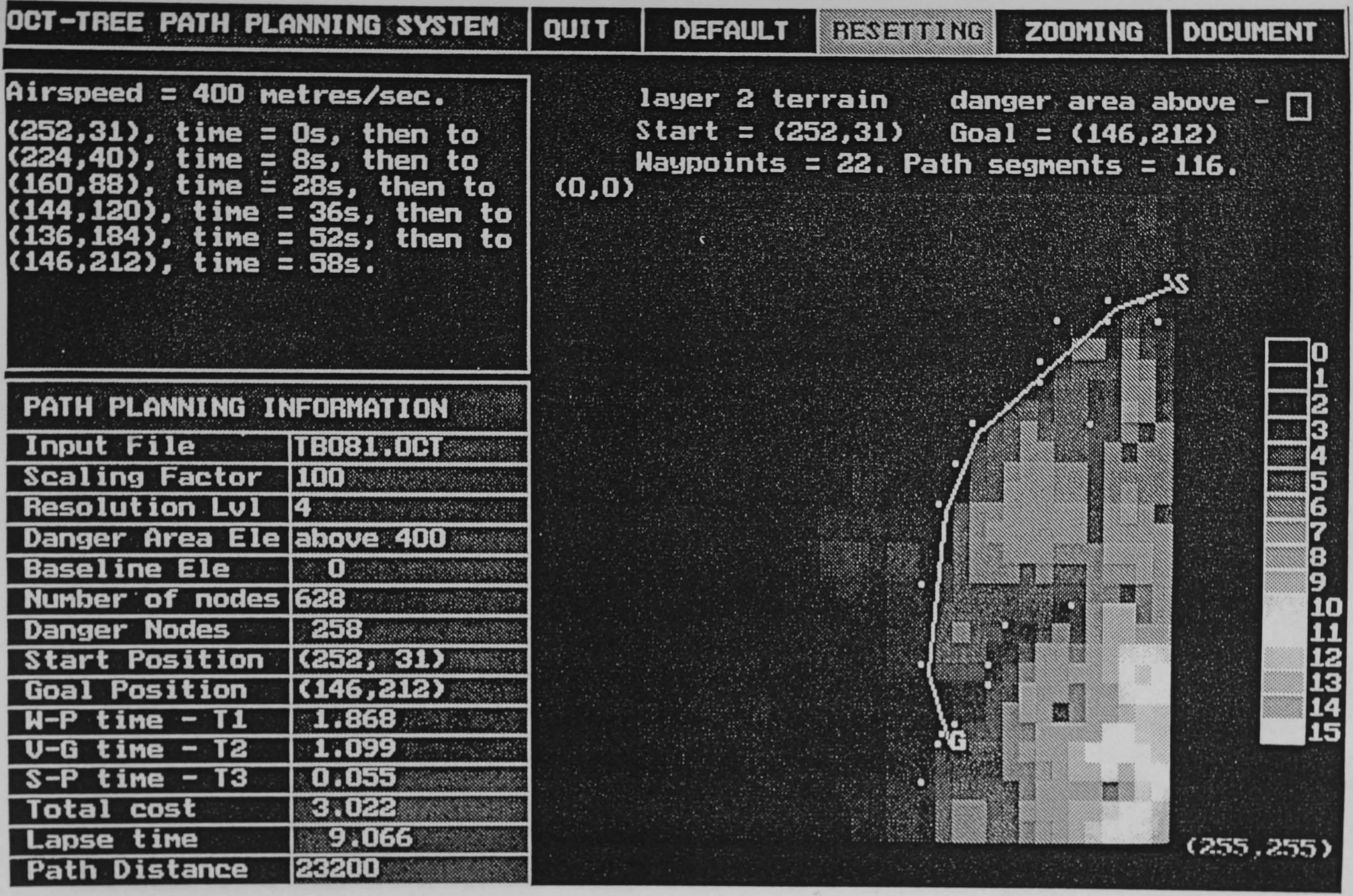


Figure 20b Path searching using the A* algorithm.

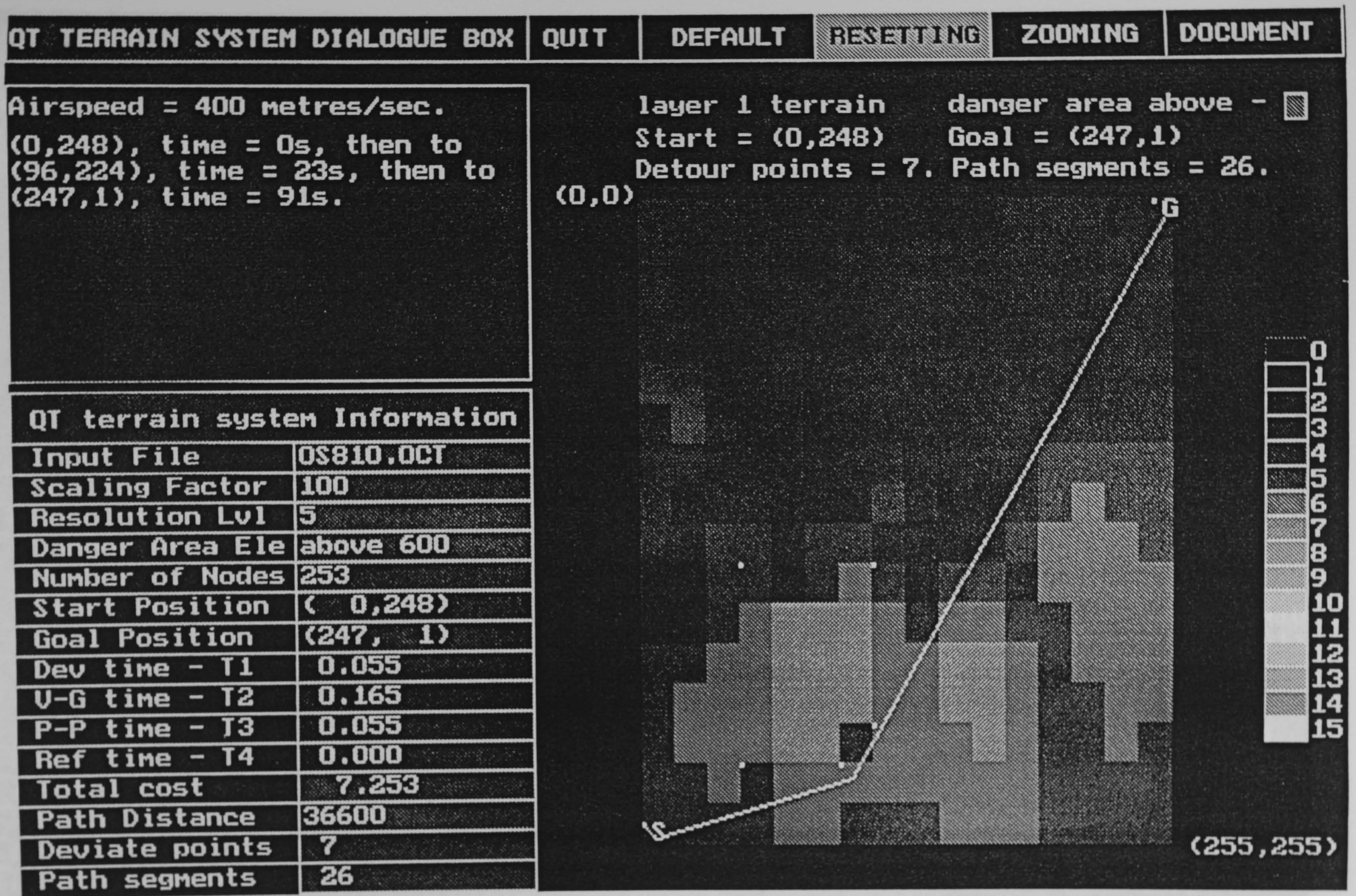


Figure 21a Path planning at layer 1 of a terrain pyramid.

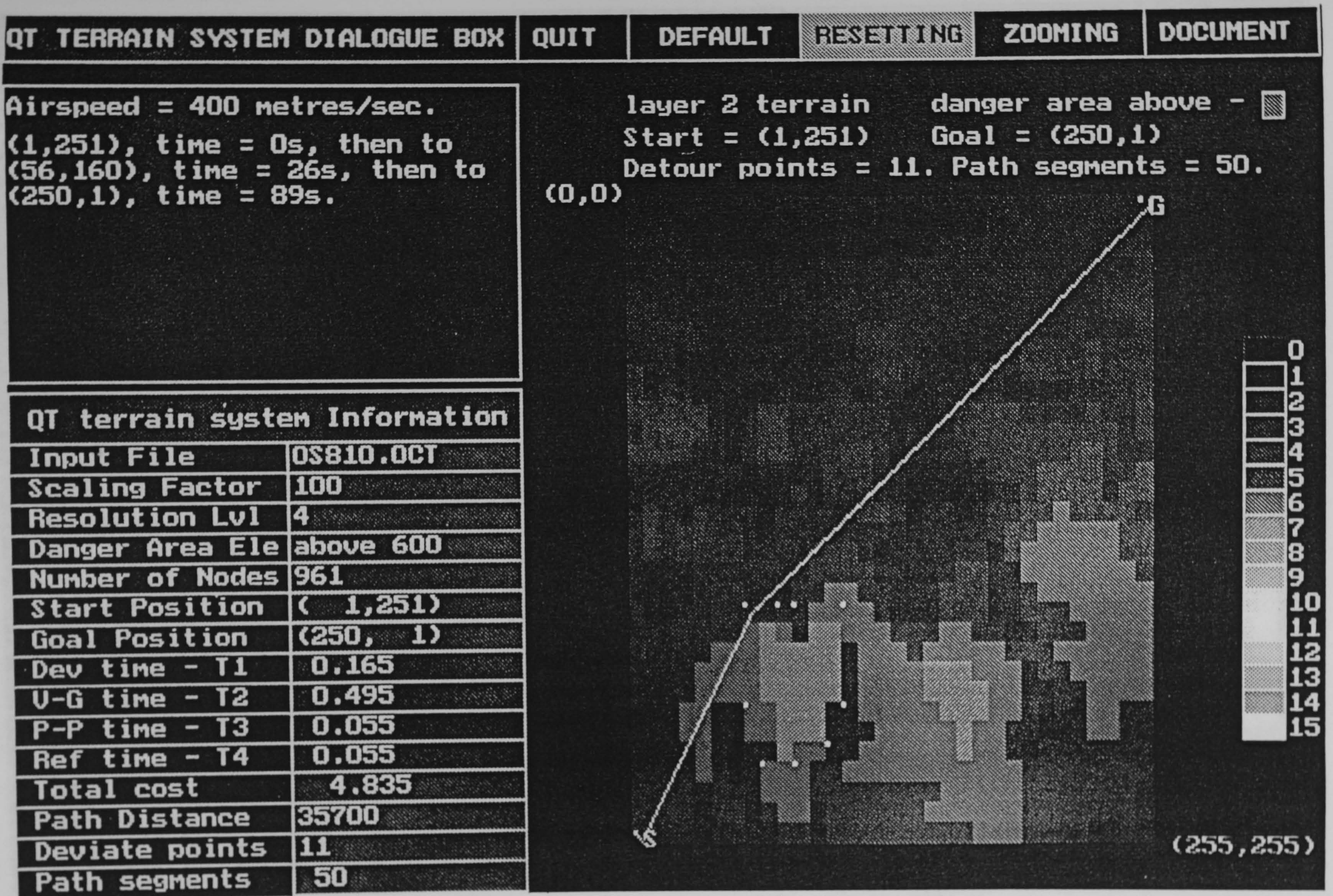


Figure 21b Path planning at layer 2 of a terrain pyramid.

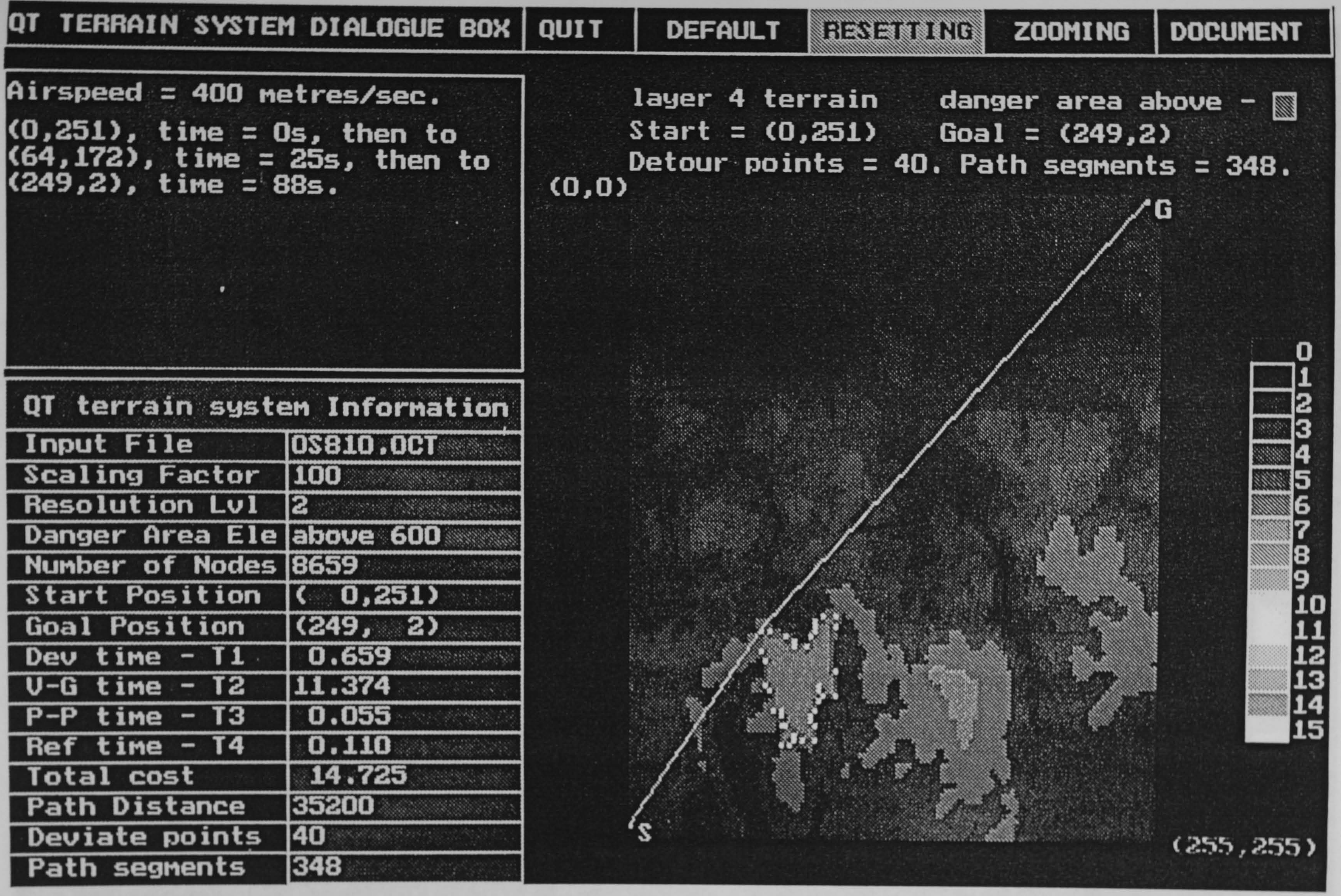


Figure 21c Path planning at layer 4 of a terrain pyramid.

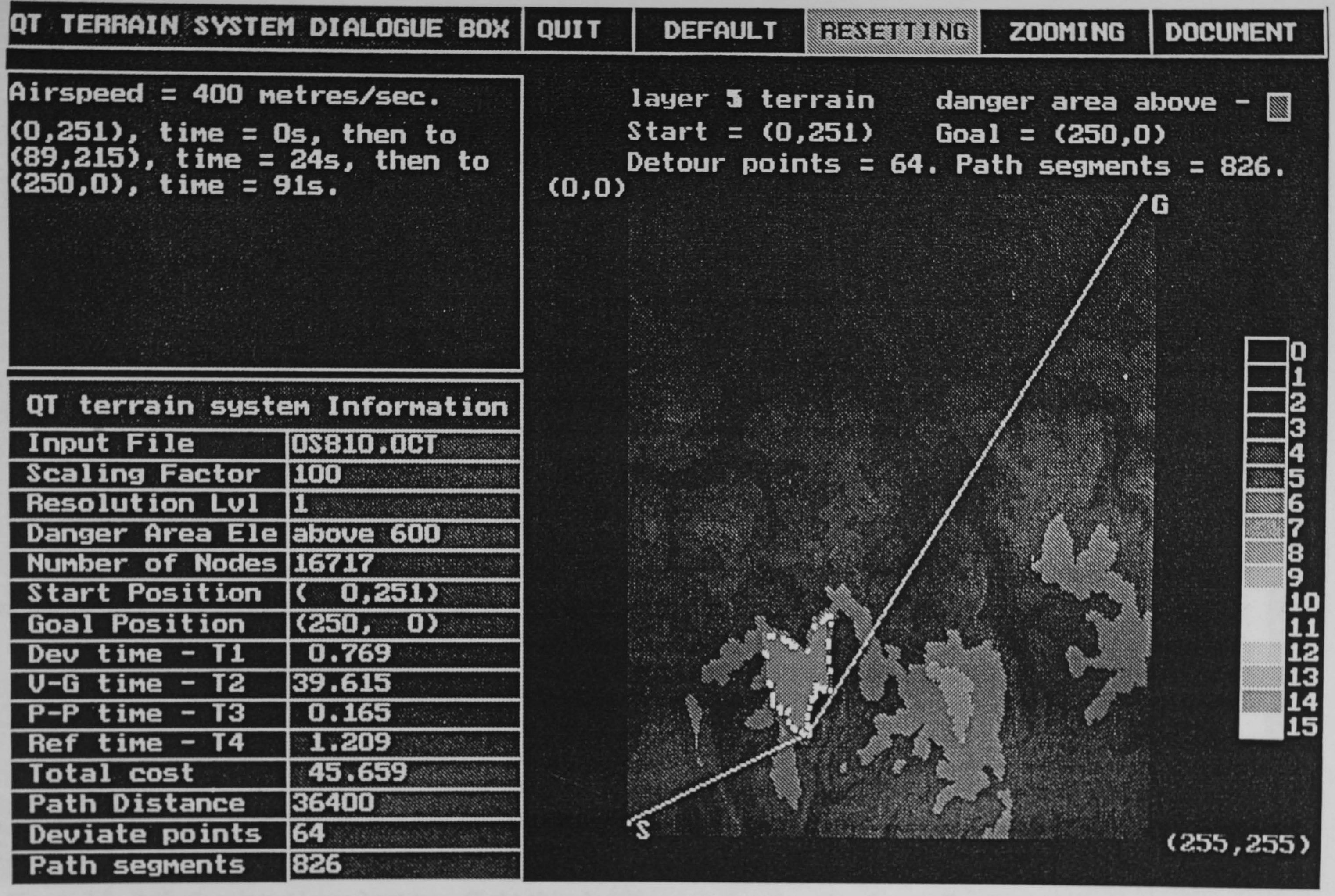
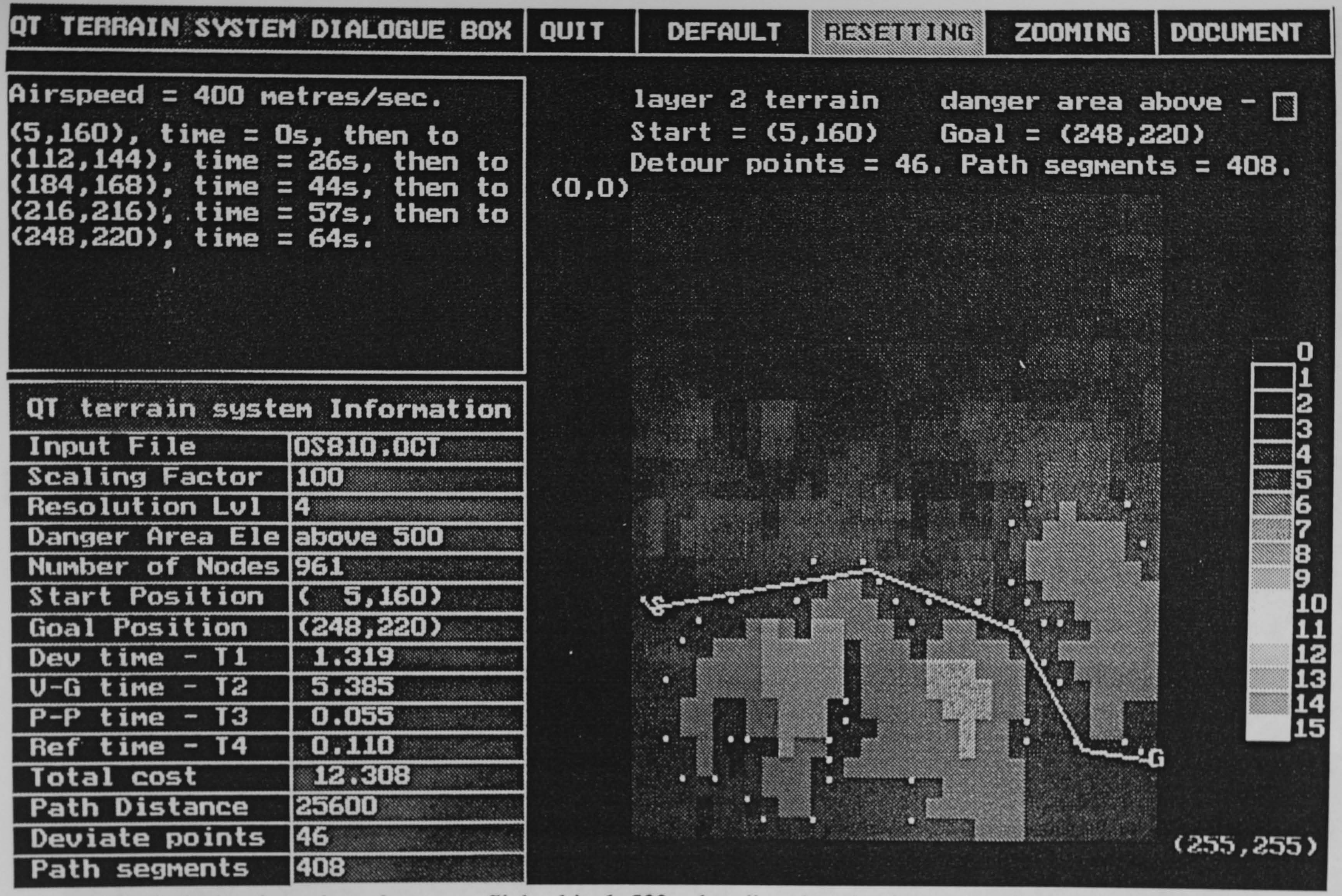


Figure 21d Path planning at layer 5 of a terrain pyramid.



QT terrain system Information	
Input File	OS810.OCT
Scaling Factor	100
Resolution Lvl	4
Danger Area Ele	above 500
Number of Nodes	961
Start Position	(5,160)
Goal Position	(248,220)
Dev time - T1	1.319
V-G time - T2	5.385
P-P time - T3	0.055
Ref time - T4	0.110
Total cost	12.308
Path Distance	25600
Deviate points	46
Path segments	408

Figure 22a Path planning at layer 2 oct-tree, flight altitude 500m, baseline elevation 0m, scaling factor 100m.

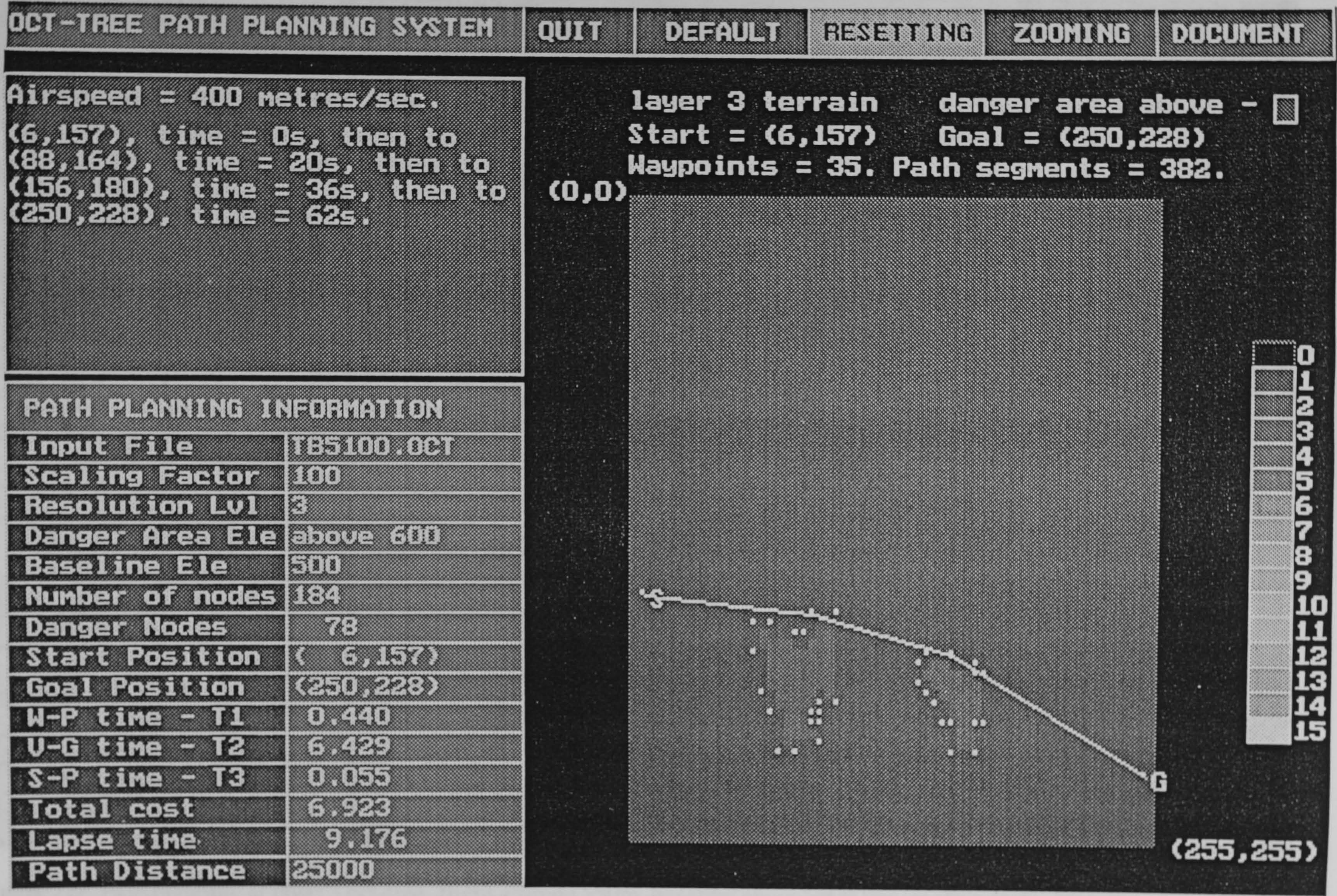


Figure 22b Path planning at layer 3 oct-tree, flight altitude 600m, baseline elevation 500m, scaling factor 100m.

OCT-TREE PATH PLANNING SYSTEM | QUIT | DEFAULT | RESETTING | ZOOMING | DOCUMENT

Airspeed = 400 metres/sec.

(17,248), time = 0s, then to
(44,176), time = 19s, then to
(170,23), time = 69s.

PATH PLANNING INFORMATION	
Input File	TB5010.OCT
Scaling Factor	10
Resolution Lvl	3
Danger Area Ele	above 600
Baseline Ele	550
Number of nodes	835
Danger Nodes	239
Start Position	(17,248)
Goal Position	(170, 23)
W-P time - T1	0.604
V-G time - T2	3.077
S-P time - T3	0.110
Total cost	3.791
Lapse time	4.011
Path Distance	27600

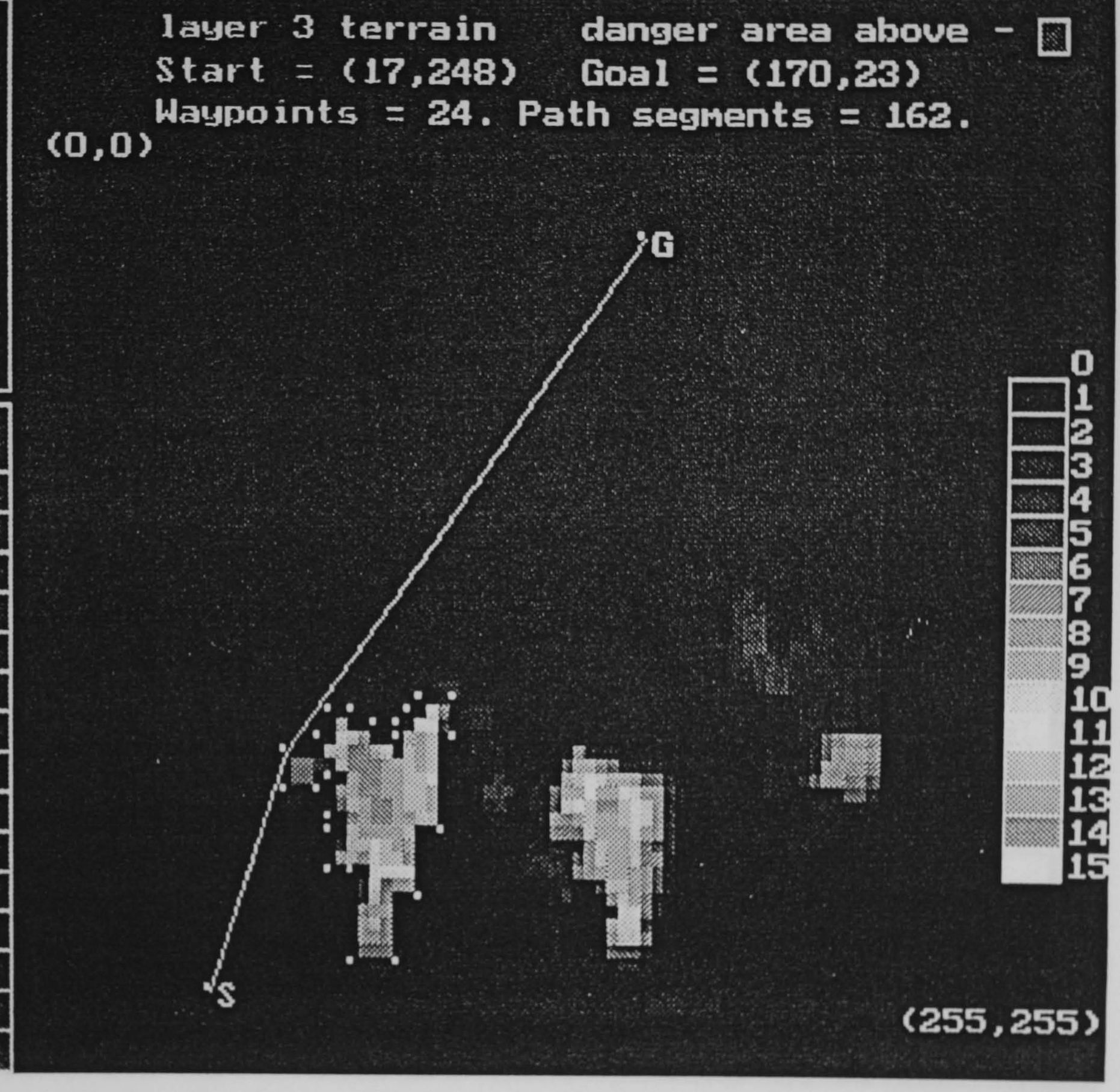


Figure 22c Path planning at layer 3 oct-tree, flight altitude 550m, baseline elevation 500m, scaling factor 10m.

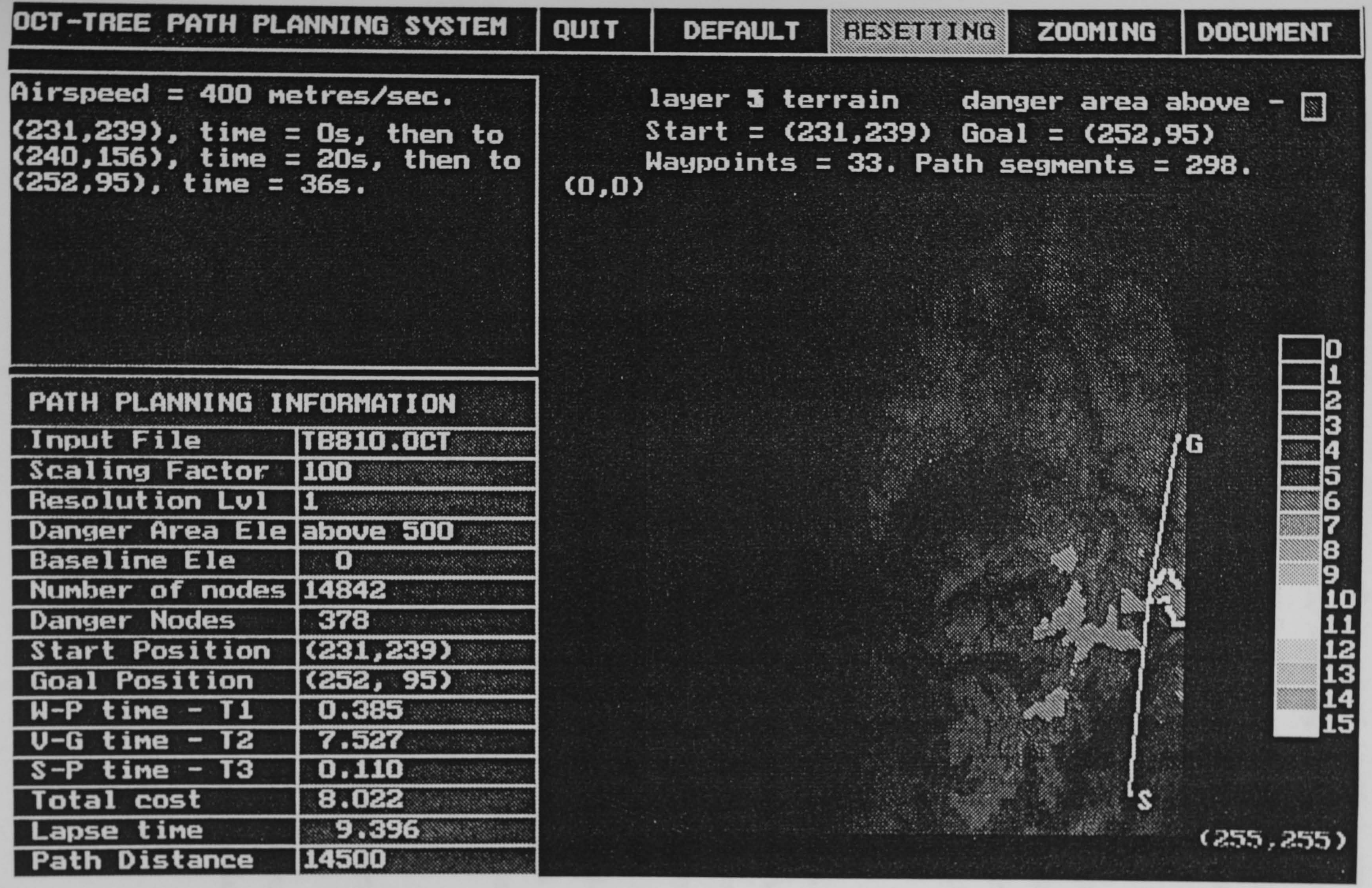


Figure 22d Path planning at layer 5 oct-tree, flight altitude 500m, baseline elevation 0m, scaling factor 100m.

During the 'flight', the user interfaces with the system using the keyboard or the mouse. The system displays messages which contain the current flight altitude, the aircraft position in co-ordinates, the current oct-tree layer, the corresponding co-ordinates and elevation of the obtained path, time to reach each waypoint along the path, time to reach the goal point, the distance of each path segment and the distance of the overall flight path.

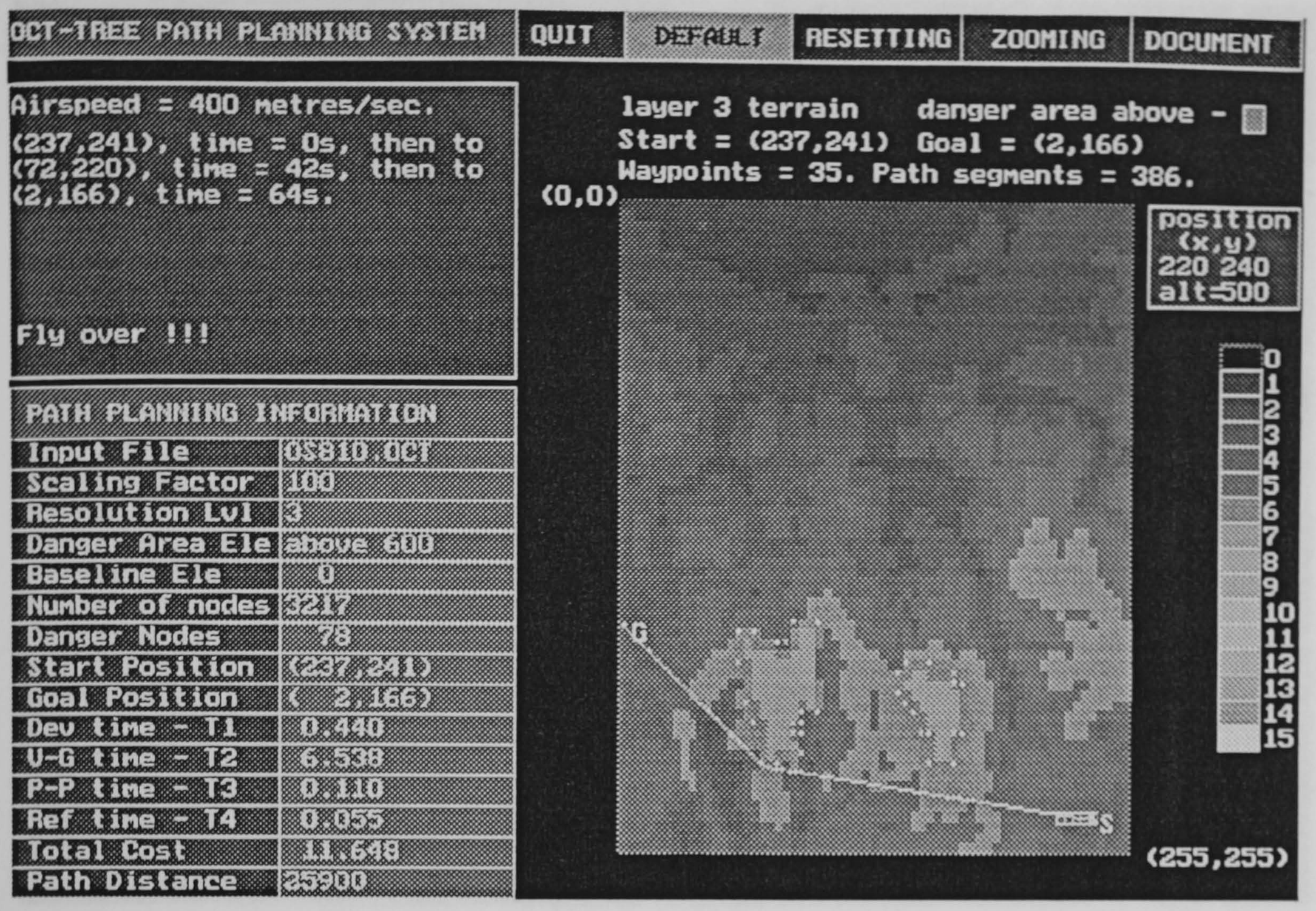
In addition to displaying the colour-coded terrain representations, the system provides a graphic interface to enable the user to issue a request for a new path and to define the overall flight conditions, for example to allow the user to define the start and goal positions either during or before each flight, and also to activate the path planning process during a flight.

As described in chapter 5, in order to meet real-time constraints, the path planning algorithm is preset at layer three where the number of waypoints is usually less than 20. However, the terrain elevation data is continuously varying and the terrain oct-tree representation is terrain dependent, thus the location and connectivity of the obstacles with respect to a given flight altitude are unpredictable. Path planning at layer 3 does not guarantee to generate a set of waypoints which can match the real-time constraints. But, as discussed in chapter 5, the cost (T_{wp}) to generate waypoints only constitutes a small amount of time in comparison with the cost of the total path planning process; this feature enables the estimation of an appropriate operation layer.

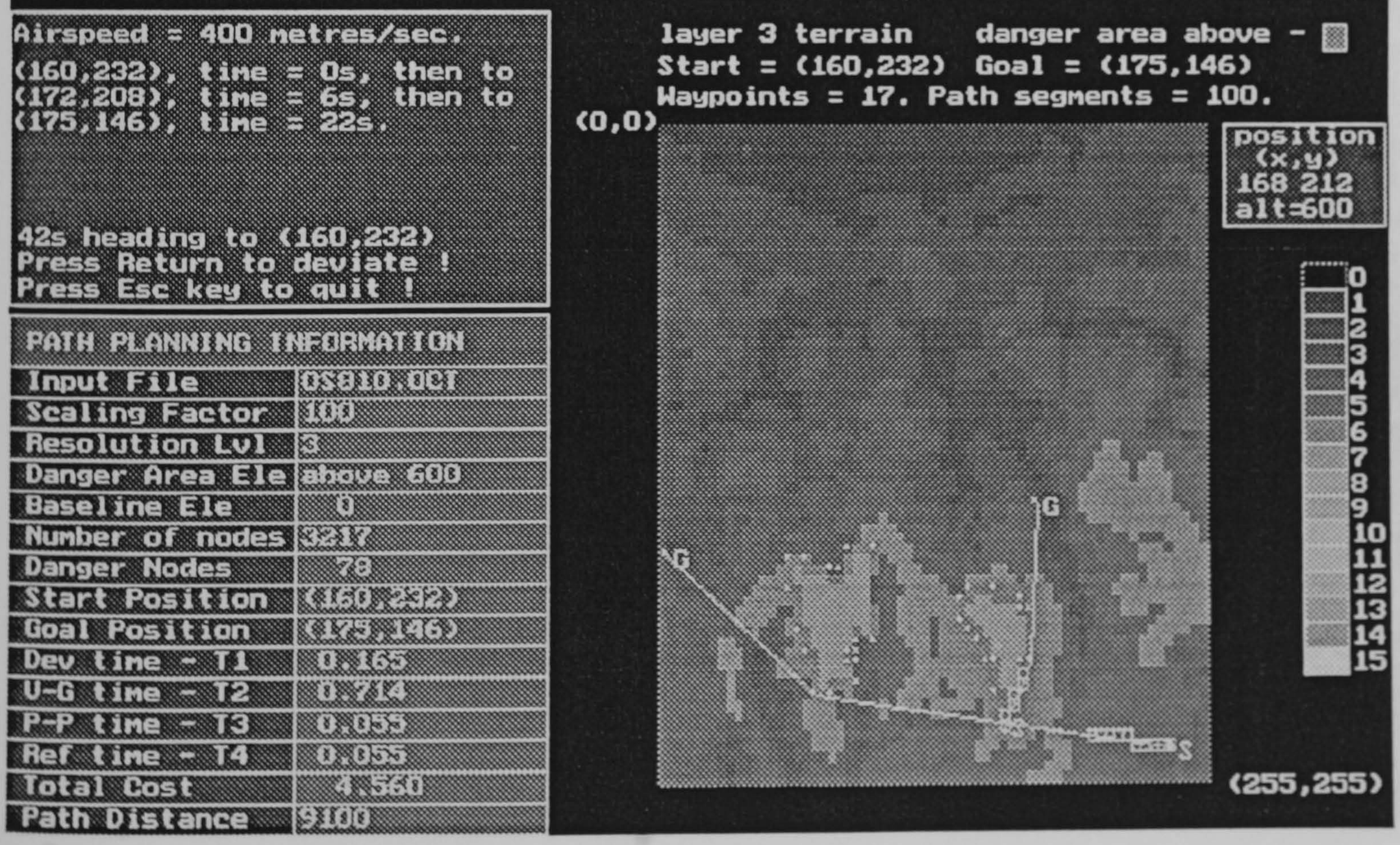
Generally, the actual operation layer of the path planning process is determined by the flight altitude which in turn determines the number of waypoints. A low flight altitude will generate a large number of danger nodes and obstacle nodes, thus a coarser operational layer may be adopted to keep the number of waypoints below a predefined margin for a specific real-time operational environment.

Figures 6.23 - 25 present examples of real-time flight path planning simulation by continuously changing the goal point during a flight mission at various flight altitudes and resolution layers, for a predefined airspeed of 400 m/sec where the flight altitude and operation layer are set interactively by the user. The results show that the shortest path can be found if the path exists in the current visibility graph; otherwise, the algorithm determines a new operation layer or alternatively selects a higher flight altitude until there are no obstacles in the navigation space.

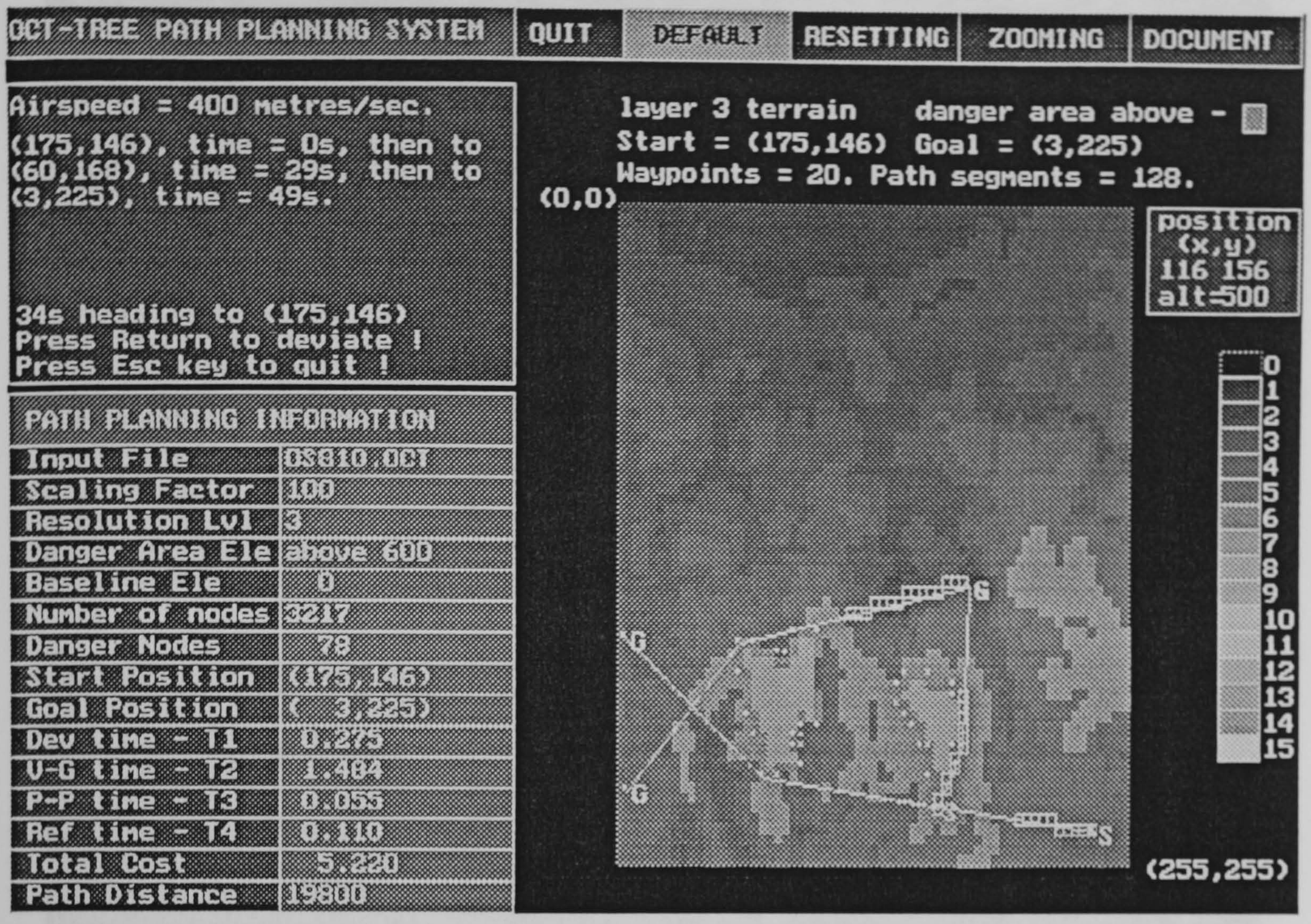
For example, Figure 6.23 demonstrates a real-time simulation in which the initial start point is (237,241) and the goal point is (2,166). During the flight, a new goal point (175,146) is given for new flight path planning; the algorithm predicts the new start point as (160,232) according to a 10-seconds constraint. The path is found in the same layer and flight altitude as shown in Figure 6.23b. New paths for various destinations are found as shown in Figure 6.23c and 6.23d. Different baselines, scaling factors, time constraints and flight altitude are applied to these simulations as shown in Figure 24 and 25.


OCT-TREE PATH PLANNING SYSTEM
QUIT
DEFAULT
RESETTING
ZOOMING
DOCUMENT
Airspeed = 400 metres/sec.
(237,241), time = 0s, then to
(72,220), time = 42s, then to
(2,166), time = 64s.
Fly over !!!
PATH PLANNING INFORMATION
Input File 05810.OCT
Scaling Factor 100
Resolution Lvl 3
Danger Area Ele above 600
Baseline Ele 0
Number of nodes 3217
Danger Nodes 78
Start Position (237,241)
Goal Position (2,166)
Dev time - T1 0.440
V-G time - T2 6.538
P-P time - T3 0.110
Ref time - T4 0.055
Total Cost 11.648
Path Distance 25900
layer 3 terrain danger area above -
Start = (237,241) Goal = (2,166)
Waypoints = 35. Path segments = 386.
(0,0)
position
(x,y)
220 240
alt=500
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
G
S
(255,255)

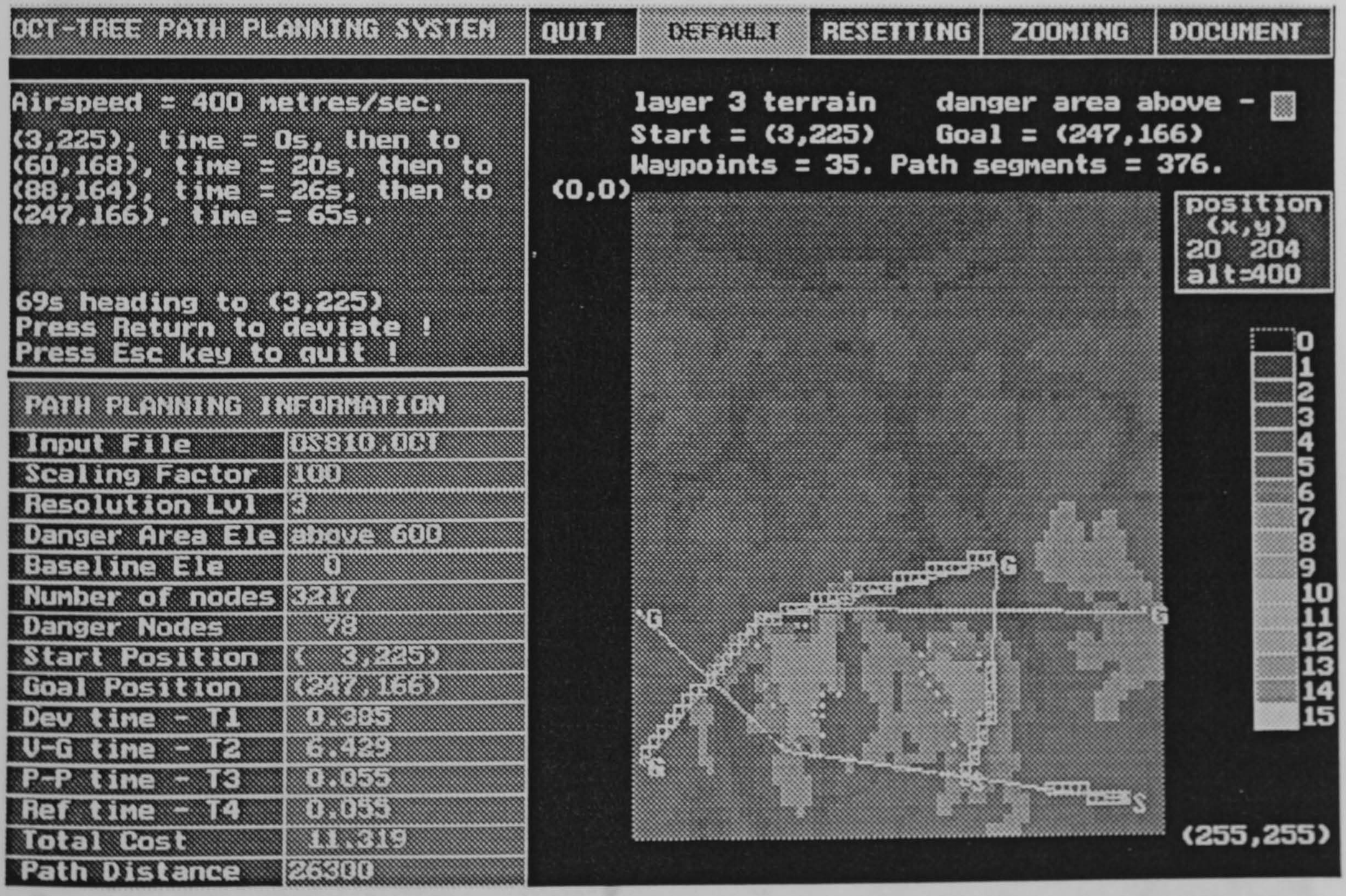
23a


OCT-TREE PATH PLANNING SYSTEM
QUIT
DEFAULT
RESETTING
ZOOMING
DOCUMENT
Airspeed = 400 metres/sec.
(160,232), time = 0s, then to
(172,208), time = 6s, then to
(175,146), time = 22s.
42s heading to (160,232)
Press Return to deviate !
Press Esc key to quit !
PATH PLANNING INFORMATION
Input File 05810.OCT
Scaling Factor 100
Resolution Lvl 3
Danger Area Ele above 600
Baseline Ele 0
Number of nodes 3217
Danger Nodes 78
Start Position (160,232)
Goal Position (175,146)
Dev time - T1 0.165
V-G time - T2 0.714
P-P time - T3 0.055
Ref time - T4 0.055
Total Cost 4.560
Path Distance 9100
layer 3 terrain danger area above -
Start = (160,232) Goal = (175,146)
Waypoints = 17. Path segments = 100.
(0,0)
position
(x,y)
168 212
alt=600
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
G
G
S
(255,255)

23b

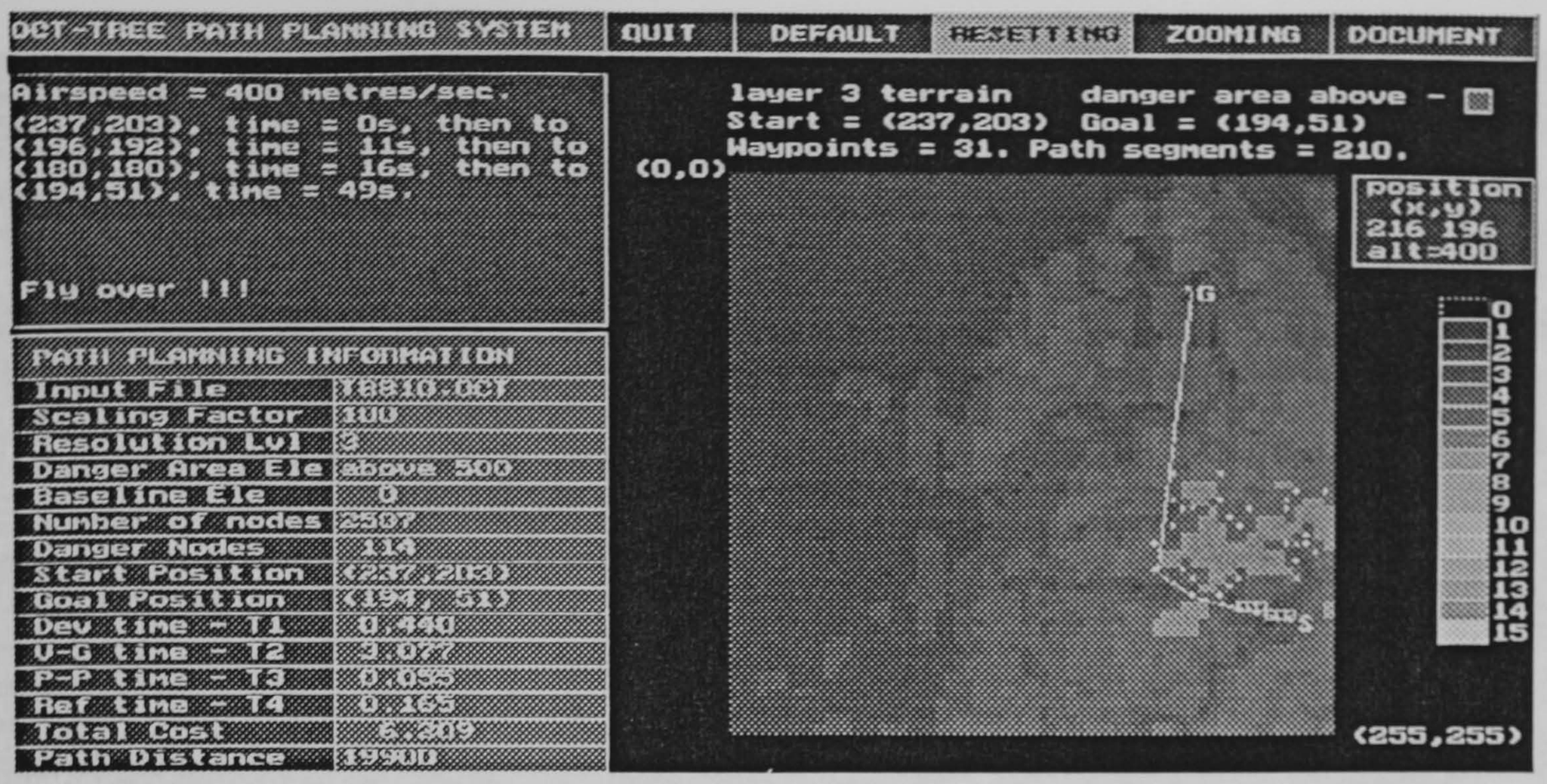


23c

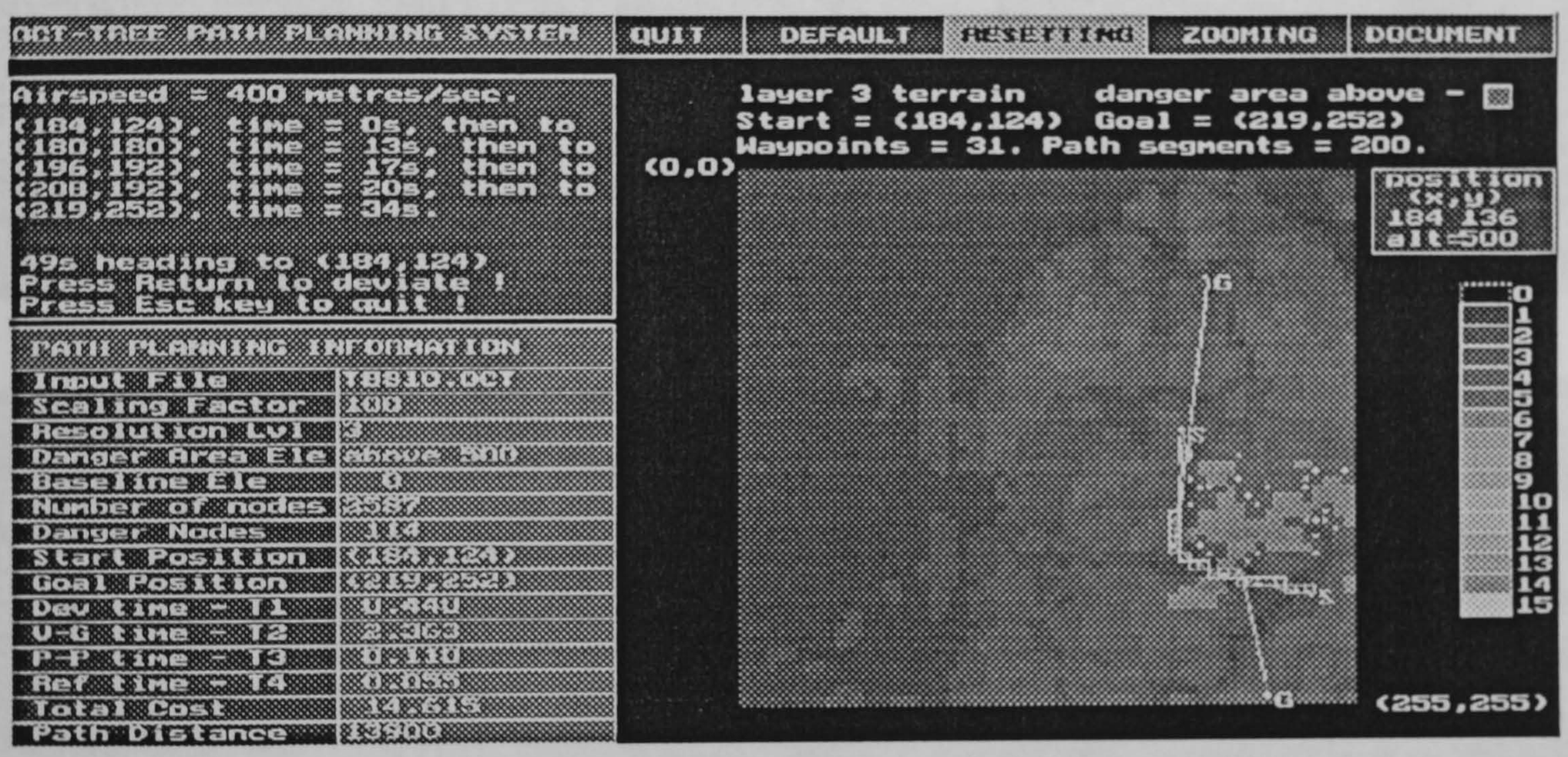


23d

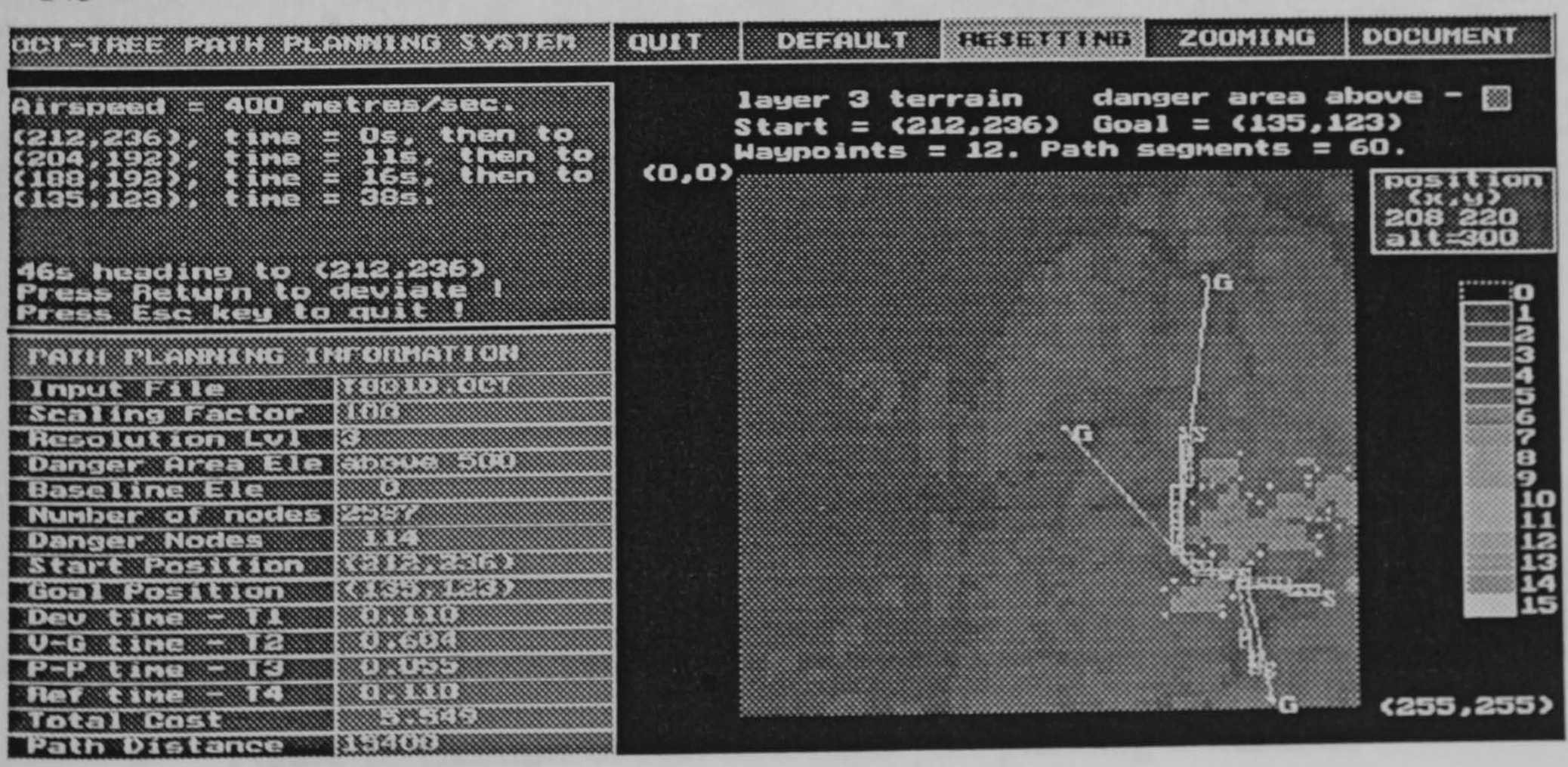
Figure 23 Step-wise real-time flight path planning simulation.



24a

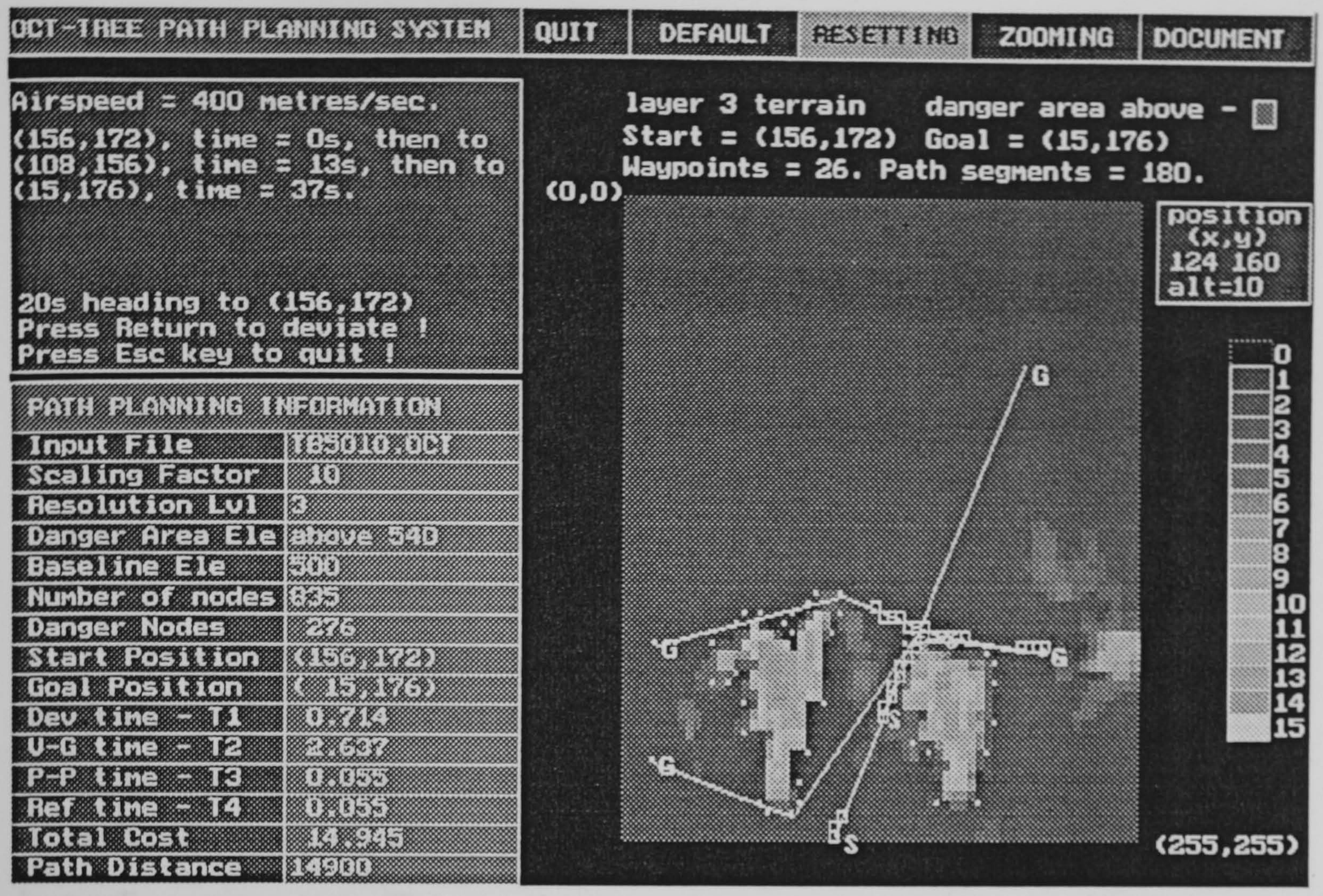


24b

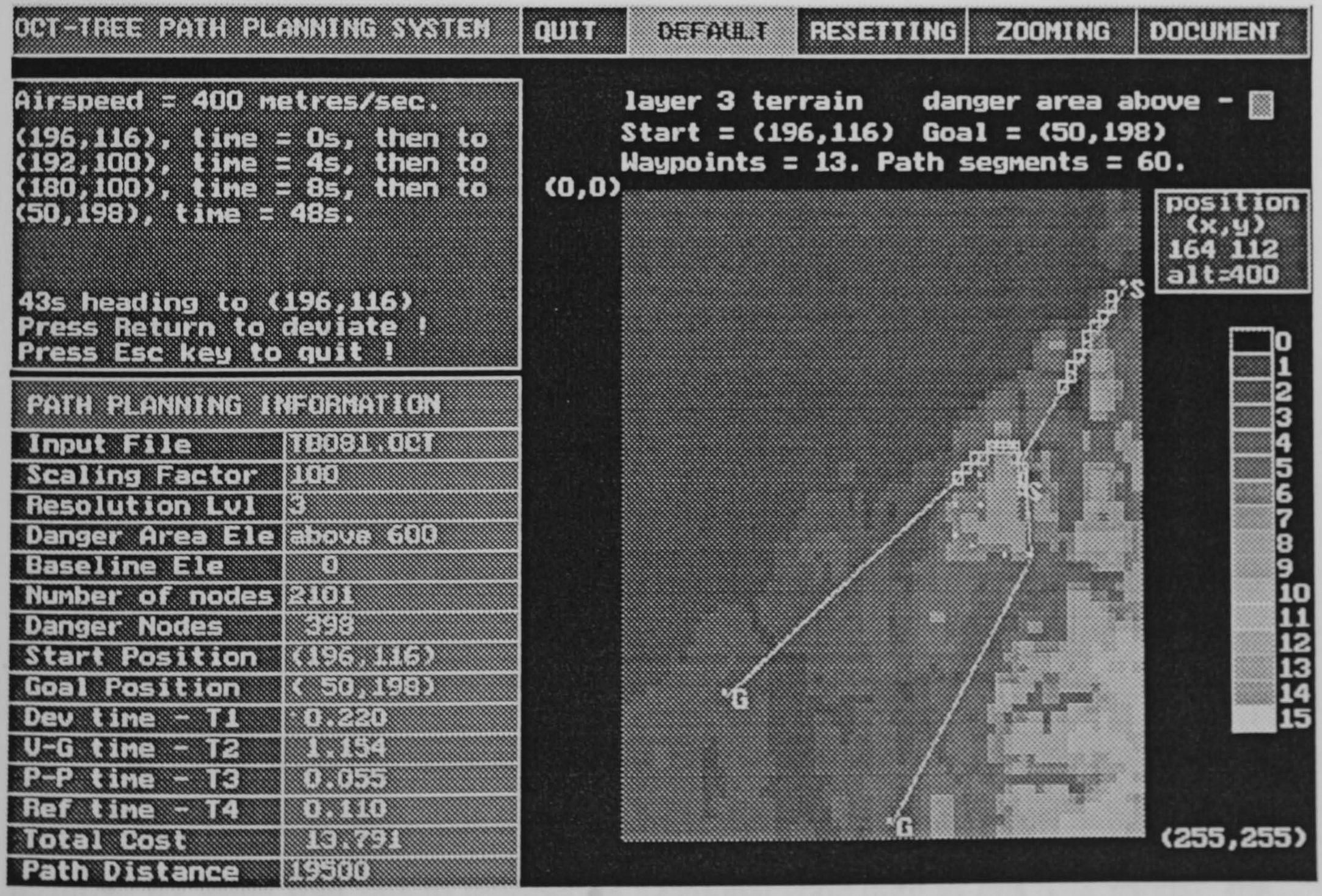


24c

Figure 24 Example of real-time flight path planning simulation.



25a



25b

Figure 25 Example of real-time flight path planning simulation.

CHAPTER 7

CONCLUSIONS

7.1 Summary of the Thesis

The initial goal in this thesis was to demonstrate the feasibility of applying a quad-tree/oct-tree data structure to digital terrain elevation data for airborne navigation. In the first part of this thesis, a data structure and encoding method have been developed which both represent and compress terrain data in an efficient way.

When the quad-tree and oct-tree structures are based on the principle of regular decomposition, by applying this principle to terrain data representation, a variant termed a terrain oct-tree is devised. The representation is based on a cell decomposition scheme in which each cell has a simple geometry, that is, it contains a scaled elevation value in a cell coverage area. The use of oct-tree data structures for terrain representation allows the geometric computation on subsets of terrain data, for example, the use of a list of nodes to represent the homogeneous elevation areas and the manipulation of a subset of nodes to provide obstacle avoidance.

Morton coding has been applied to the ordering of both quad-tree nodes and oct-tree nodes. The vertical downward projection of any oct-tree nodes has the same locational code as an equivalent quad-tree and furthermore the derivation of these projection codes reduces to simple logical operations. Moreover, the relationship between two-dimensional and three-dimensional locational codes simplifies access operations on a terrain by allow most of these operations to be executed in two-dimension space.

This new encoding method represents the spatial point of an elevation peak in three-dimension locational codes while retaining the conventional quad-tree and oct-tree features. By using four bits to represent a locational code digit in this data format, the

method does not require any extra storage in comparison with the standard linear oct-tree encoding method.

In the second part of this thesis, the terrain oct-tree model is then applied to real time flight path planning. There are several terrain navigation functions such as obstacle avoidance, terrain following, observability and terrain matching which are closely related to the flight path and the topology of a region of terrain. The flight path planning application was selected as an example to demonstrate that the oct-tree based terrain model has the potential to be used in airborne navigation.

Both the variable resolutions and multi-resolutions of a hierarchical data structure have been implemented. The variable resolution features of quad-trees are used to approximate the obstacle nodes and to locate the vertices of the obstacle region for navigation space representation, where the multi-resolutions feature of a pyramid is used to provide a compact and proper navigation space for real time application of flight path planning.

Attention was concentrated on the implementation of the terrain oct-tree to described navigation space. The operations on the terrain oct-trees, specifically the accessing and retrieval of terrain data and the modelling of searching space have been demonstrated. The constraints put on the flight path planning method include the minimum flight altitude, a time limit to obtain a preferable path and aircraft speed. The real-time implementation is performed using time constraints which are derived from the experimental results of the performance of the algorithm in the static environment.

7.2 Conclusions

In this thesis, a technique has been developed for the efficient storage of terrain elevation data represented by the leaf nodes of an oct-tree hierarchical data structure. There are two major advantages of the terrain oct-tree scheme. Firstly, space efficiency is improved in comparison with both the DTED file and standard oct-tree encoding

methods. Secondly, manipulation of the application process to allow three dimensional problems to be manipulated in two dimension space.

The ability to exploit redundancy and to offer flexibility to represent terrain at coarser levels of DTED is provided by a hierarchical terrain representation. Oct-trees offer a method to compress digital terrain elevation data. Common regions of terrain are merged within the tree and most importantly, as the oct-tree structure enables a terrain to be represented in different degrees of resolution, there is sufficient information at any node within the tree to determine if the terrain data at that level of the tree is adequate for a specific navigation application.

For example, the encoded terrain oct-tree is amenable to navigation algorithms which utilise variable and multiple resolutions of terrain, where the resolution is a function of the number of elevation elements. Thus a coarser resolution representation of terrain has less nodes but with a larger coverage area per node. As the resolution decreases, the fine detail reduces.

It is clear from the experimental results that the number of nodes can be controlled by the scaling factor, the baseline elevation and the layer in the oct-tree pyramid. A larger scaling factor results in a smaller number of nodes but the elevation error in the tree is increased with the scaling factor. If a higher baseline elevation is used during the encoding process, the number of nodes also reduces and attention can be given to a specific range of elevation by using a smaller scaling factor without losing accuracy. The implementation of a pyramid allows a global change of resolution and further reduction in the number of nodes in the upper layers of a pyramid.

Although the terrain oct-tree structure introduces rounding errors in scaled elevation, the margin of error is limited by the scaling factor. The retrieved elevation is always higher than the corresponding true elevation to ensure flight safety, besides, the locational codes of a node can always be used as an index to retrieve the true elevation data in the original DTED file and to ensure there is no position error in a

horizontal direction.

In Cebrian's [Cebr 84] linear quad-tree terrain model, the full matrix data is integrated into a linear quad-tree. It uses the same number of pixels as a raster image to represent a region. It stores all the lowest leaves of a quad-tree but loses the hierarchical properties of a quad-tree and introduces redundant information. Chen's [Chen 86] quad-tree terrain model uses two numbers, an address value in Morton numbering sequence and associated elevation data, to represent a single node in a quad-tree. The elevation data is obtained from the recursive approximation of four sibling quadrants. A predefined error bound determines whether the subdivision should proceed and results in a local approximation of elevation data.

In the terrain oct-tree model proposed in this thesis, the scaling function is a global process applied to the entire terrain elevation data. Only one integer number (locational code) is necessary to represent a single node which explicitly contains the spatial position, scaled elevation and the size information of a node.

The time performance of accessing a node in a terrain oct-tree is $O(logN_t)$ which is dependent on the number of nodes in the tree. The operations include the transformation of the co-ordinates to the locational codes, the binary search and the retrieval of the three dimensional coordinates (as well as the size information) of a node.

In demonstration of the terrain oct-tree implementation, an approach has been proposed to solve the path planning problem for an aircraft within an environment where only the terrain elevation data is given. By including time as a constraint in the planning process, a new real-time flight path planning algorithm was developed. The minimum flight altitude is used to model the obstacles configuration in the navigation space. The fundamental factors in aircraft navigation, that is, flight altitude, speed and time are provided and conditions such as optimization of the flight path with respect to flight distance and detour points are imposed on a flight path in the search procedure.

Due to the one-to-one mapping between the two-dimensional and the three-dimensional locational codes, most parts of the path planning algorithm are performed in quad-tree space. The integer data format of the locational codes lends itself to manipulation by logic operations. The data components (three dimensional coordinates and size) of a node can be processed individually without interfering with the other components of the node. This data format improves the time performance as well as the storage requirement during the process.

Collision checking in the path planning process affords improvements in speed by exploiting the variable and multiple resolution characteristics for terrain oct-tree based applications. For example, collision checking is performed by way of a search against a danger nodes list without actually locating the geometric position of the danger areas in navigation space and avoids costly geometric computations. As the number of nodes in the list depends on the resolution and the minimum flight altitude, the oct-tree structure facilitates fast generation of a new danger nodes list at a coarser resolution level.

Once a node is identified as an obstacle, locational codes of the elevation element can be ignored. This strategy simplifies the domain of operation for two-dimensional and binary representations where a polygon in navigation space is either free space or an obstacle. This approach can be extended to other applications such as the introduction of terrain features or threaten overlaying and extraction.

Most digital terrain models adopt a grid graph of free space for path searching and most robot motion planning problems use visibility graphs or Voronoi graphs to represent the search space, where the polygonal obstacles are given by their geometric representation. The flight path planning algorithm developed in this thesis has the feature that the visibility graph is used in a digital terrain model to represent the search space to gain the benefits of space and time efficiency for path searching without reference to pre-defined obstacle information in the navigation space. In the method

proposed in this thesis, the obstacles need to be explored dynamically during navigation.

However, the bottleneck in the time complexity for the visibility graph approach remains. The algorithm needs to determine, for every pair of waypoints, if the waypoints are visible to each other. The visibility graph construction method in this algorithm has to check every pair of vertices against the danger nodes list with a performance of $O(W^2)$. Although the visibility graph construction time cost is reduced by selecting the appropriate resolution layer to reduce the number of waypoints by using a binary search on the danger nodes list and by only constructing the partial visibility graph of the navigation space in line with the total cost of the flight path planning algorithm, the method is still expensive.

For long range global path planning in an airborne environment, the oct-tree hierarchical structure provides the advantage of avoiding excess detail on parts of the space that do not affect the planning operations. If the process is conducted at a resolution which is l levels coarser than the pixel level, the checking for collision, the number of danger nodes and the number of nodes along a path segment are reduced by a factor of 2^l, substantially reducing the time complexity of the search.

The flight path planning approach is able to limit the size of searching space by establishing a partial visibility graph of the navigation space, and is capable of avoiding detail that does not affect the selection of the path, independent of the size of the navigation space.

In a search graph, a single node may be reached from the start node by different paths with different costs. Once the searching space is defined, the remaining problem is the implementation of effective searching methods. The cost of the search space can be attached to each path segment during the construction of a visibility graph for optimal path searching. However, in this thesis, the cost function was restricted the flight distance only.

In summary, this research has following achievements:

1. This thesis demonstrates the applicability of a novel terrain oct-tree structure to aircraft navigation. The terrain oct-tree structure provides a compact terrain representation not only to overcome the natural disadvantages of DTED but also to improve operational performance without significant loss of data accuracy or flight safety. Empirical tests indicate that this data structure performs well both in terms of storage efficiency and data manipulation.

2. The terrain oct-tree representation retains the original features of standard quad-tree and oct-tree structure. The encoding method proposed in this thesis can be extended to a multiple-attributes representation where the standard quad-tree (pointer and linear quad-tree) structure is only applied to binary image representation.

3. From the path planning stand-point, the terrain oct-tree afford a real-time operational capability. The path planning strategy avoids costly geometric computations and is straightforward to implement. More importantly, the path planning approach is a combination of a digital terrain representation and a geometric navigation space configuration for path searching.

4. The terrain oct-tree provides a systematic approach to reorganising terrain elevation data. The terrain oct-tree can be used as an alternative data base for a terrain referenced navigation system, particularly, when the elevation data is too large to be handled in the existing DTED file.

7.3 Future Research

Terrain referenced navigation is a relatively new technique for updating inertial navigation systems in manned aircraft. One of the key problems in TRN systems is the

management of large volumes of DTED data for various navigation applications. The advanced avionics, flight control algorithms, route and mission planning, and evaluation of low-altitude tactical and strategic missions are critically dependent upon real-time terrain data access.

The terrain oct-tree may be used as a source of data for alternative navigation applications that make use of DTED data. Some of these navigation functions are directly related to the flight path planning algorithm covered in this thesis:

- Mission Planning Integration - A mission planning system would provide the user with all the information required to plan, printout, display and prepare the mission data 'up-load' for the aircraft. Other functions such as radar profile predictions, terrain masking, visual simulation, and pre-mission simulation may also be created from terrain data and integrated into the system. By using a terrain oct-tree database in lieu of DTED to develop such a system, the benefit of using a hierarchical data structure can be extended to the data transmission and cockpit display generation.

- Terrain Following/Terrain Avoidance - By applying the flight path planning algorithm iteratively to a preplanned global reference flight path, the local trajectory can be improved by minimizing a combination of flight altitude and deviation from the preplanned path subject to a prescribed flight path and manoeuvring constraints.

  As the flight path planning algorithm provides a long range flight path ahead of an aircraft, the elevation data along the path can be used to generate a synthetic profile at any horizontal resolution as required for terrain following flight. Since each elevation value along a path is a maximum value approximated within a node coverage area, the safety in clearance is always guaranteed. Ground and Obstacle Collision Avoidance can be performed at an early stage of a flight path under the terrain oct-tree representations.

- The Obstruction Database - Database updates are likely to be infrequent for DTED data whereas the obstruction database is likely to change more often and is much smaller than a DTED. The use of an obstruction database based on locational codes improves the storage and time performance of some applications. Since the database is a linear list of locational codes, update the database is straightforward. An arbitrary polygonal obstacle can be derived directly by converting the vertices of a polygonal, and the boundary representation of a polygonal into a quad-tree representation.

- Threat Processing - Threat information is used along with the digital terrain to compute the intervisibility polygons. The threat can be represented in locational codes which allows threats to be added, removed, modified, saved in a file, loaded from a file, and drawn as overlays. Moreover, the method of generation of a 'danger index' for each path segment in navigation space needs to be explored in a mission planning system if the system is to make use of the terrain oct-tree representations.

- Line of Sight (LOS) - Finding terrain altitude and determining lines of sight over polygonal database is computationally expensive. This is because the general solution for finding terrain height requires a comprehensive search. Every polygon in the database must be tested to determine if it lies above the spatial position where terrain height is to be calculated.

  Determining if a clear line of sight exists between two points requires a similar search. The polygon database can be simplified by encoding it into locational codes of terrain oct-tree. The collision checking technique developed in chapter four thus can be applied to determine the LOS. This process can be further extended to maintaining minimum or maximum LOS visibility from a known position within the terrain.

- Digital Map Displays - In airborne operation, the aircraft altitude, heading, location and selected map modes are integrated to build the digital map displays. At present these displays are based on digitised paper charts stored in the form of digitised images where DTED data is used to enhance the display. By using the terrain oct-tree, the digital maps can be displayed in variable or multiple resolutions, For example maps can be displayed in a resolution as a function of coverage area and altitude, or to show areas above the current aircraft height, for flight safety purposes.

- Terrain Matching - The goal of the terrain matching is to match sensory input (optical, radar) to 3-D terrain data. In other words, to locate the observed terrain within the overall terrain. A measured terrain elevation profile based on 3-D locational codes provides information which is correlated with a terrain oct-tree. The measured terrain elevation profile is compared with the terrain oct-tree by using the neighbouring finding technique to locate a successive sequence of nodes with elevation and size data matches for the measured profile.

As the number of nodes in a terrain oct-tree can be reduced by changing the resolution in both the horizontal and vertical direction, unrelevant nodes can be eliminated by choosing an appropriate baseline elevation in the navigation area. The size of search space is reduced in comparison with the size of the corresponding DTED. Thus, the terrain oct-tree offers a compact representation for terrain matching.

REFERENCES

[Abel83] Abel, D.J., Smith, J.L., A Data Structure and Query Algorithm Based on a Linear Key for a Rectangle Retrieval Problem, *Computer Vision, Graphics, and Image Processing 24*, 1 Oct 1983, 1-13.

[Abel84] Abel, D.J., A B^+-tree Structure for Large Quadtrees, *Computer Vision, Graphics, and Image Processing 27*, 1., July 1984, 19-31.

[Abel85] Abel, D.J., Some Elemental Operations on Linear Quadtrees for Geographic Information Systems, *Computer Journal 28,1.*, Feb 1985,73-77.

[Aho74] Aho,A.V.,Hopcroft, J.E., Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Alle92] Allerton D.J., Gia, M.C., The Application of Oct-trees in Airborne Terrain Guidance Systems, *RIN & DGON Digital Mapping and Navigation Conference*, London, Nov 1992, paper No.11.

[Bair90] Baird, C.A., Snyder, F.B., Terrain-aided Altitude Computations on the AFTI/F-16, *IEEE Position, Location and Navigation Symposium* 1990, 474-481.

[Barf93] Barfield, F., Probert, J., Browning, D., All Terrain Ground Collision Avoidance and Maneuvering Terrain Following for Automated Low Level Night Attack, *IEEE AES Systems Magazine*, March 1993, 40-47.

[Barn84] Barney, G., Enhanced Terrain Masked Penetration Program, *IEEE National Aerospace and Electronics Conference* 1984, 90-96.

[Benn88] Bennett, P.J., Enhanced Navigation and Display From Passive Terrain Reference Avionics, *IEEE Position Location and Navigation Symposium*, Nov 1988, 209-216.

[Bert86] Berthiaume, R., Karnavas, G., Bernsteen, S., Graphical representations of DMA Digital Terrain Data on Low Cost Commercial Graphics Workstation, *IEEE National Aerospace and Electronics Conference* 1986, 992-996.

[Boys86] Boys, R.M., Terrain-Based Information: A Reason to integrate, *IEEE National Aerospace and Electronics Conference* 1986, 888-893.

[Broo83] Brook, R.A., Solving the Find-Path Problem by Good Representation of Free Space, *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(3), 1983, 190-197.

[Burn84] Burnham, G.O., Integrated Terrain Access/Retrieval System, *IEEE National Aerospace and Electronics Conference* 1984, 97-105.

[Burn87] Burnham, G.O., Smoth, S.A., Davis, K.L., Cockpit Avionics - Charting the Course for Mission Success, *IEEE National Aerospace and Electronics Conference* 1987, 97-100.

[Camb85] Cambron, T.M., Snyder, F.B., Fellerhoff, J.R., Implementation of The SITAN Algorithm in The Digital Terrain Management and Display System. *IEEE National Aerospace and Electronics Conference* 1985,78-86.

[Cann76] Canner, W.H.P., *Air Navigation*, Brown, Son & Ferguson, Glasgow, 1976.

[Cebr85] Cebrian, J.A., Mower J.E., Mark, D.M., Analysis and Display of Digital Elevation Models within a Quadtree-based GIS, *Auto-Carto 7 Proceedings*, Washington D. C., March 1985, 55-64.

[Chan85] Chan, Y.K., Foddy, M., Real Time Optimal Flight Path Generation by Storage of Massive Data Bases, *IEEE National Aerospace and Electronics Conference* 1985, 516-521.

[Chen86] Chen, Z.T., Tobler, W.R., Quadtree Representations of Digital Terrain, *Proceedings of Auto-Carto London, vol. 1*, London, September 1986, 475-484.

[Chen88] Chen, H.H., Huang, T.S., A Survey of Construction and Manipulation of Octrees, *Computer Vision, Graphics, and Image Processing 43, 3.*, 1988, 409-431.

[Come79] Comer, D., The Ubiquitous B-tree, *ACM Computing Survey 11,2.*, June 1979, 121-37.

[Cree86] Creel, E.E., Fellerhoff, J.R., Data compression Techniques for Use with the SITAN Algorithm, *IEEE Position Location and Navigation Symposium*, Nov 1986, 309-315.

[Dent84] Denton, R.V., Froeberg, P.L., Applications of Artificial Intelligence in Automated Route Planning, *SPIE vol. 485 Applications of Artificial Intelligence* 1984, 126-132.

[Dent85] Denton, R.V., Jone, J.E., Froeberg, P.L., Demonstration of An Innovative Technique for Terrain Following/Terrain Avoidance - The Dynapath Algorithm, *IEEE National Aerospace and Electronics Conference* 1985, 522-529.

[Dijk59] Dijkstra, E.W., A note on Two Problems in Connexion with Graphs, *Numerische Mathematik 1*, 1959, 269-271.

[Dyer80] Dyer, C.R., Rosenfeld, A., Samet, H., Region Representation: Boundary Codes From Quadtrees, *Communication of the ACM 23, 3.*, March 1980, 171-179.

[Dyer82] Dyer, C.R., The Space Efficiency of Quadtrees, *Computer Graphics and Image Processing 19, 4.*, August 1982, 335-348.

[Fabb86] Fabbrini, I., Montani, C., Autumnal Quadtree, *The Computer Journal, 29, 5.*, 1986, 472-474.

[Fair91] Mission Support System Overview , A Total solution to Mission Planning Requirements, Fairchild Defense, Oct 1991.

[Fox86] Fox, T.A., Clark, P.D., Development of Computer-Generated Imagery for A Low-Cost Real-Time Terrain Imaging System, *IEEE National Aerospace and Electronics Conference* 1986, 986-991.

[Fred84] Fredman, M., Tarjan, R., Fibonacci Heaps and Their Uses in Improved Network Optimization algorithms, *Proceedings 25th Annual IEEE Symposium on Foundations of Computer Science,* 1984, 338-346.

[Fuji89] Fujimura, K., Samet, H., A Hierarchical Strategy for Path Planning Among Moving Obstacles, *IEEE Trans on Robots and Automation, vol. 5, No.1.*, Feb 1989, 61-69.

[Garg82a] Gargantini, I., An Effective Way to Represent Quadtrees, *Communications of the ACM 25, 12.*, December 1982, 905-910.

[Garg82b] Gargantini, I., Detection of Connectivity for Regions by Linear Quadtrees, *Computers and Mathematics with Applications 8,4.*, 1982, 319-327.

[Garg82c] Gargantini, I., Linear Octtrees for Fast Processing of Three-dimensional objects, *Computer Graphics and Image Processing 20, 4.*, December 1982, 365-374.

[Garg83] Gargantine, I., Translation, Rotation, and Superposition of Linear Quadtrees, *International Journal of Man-Machine studies 18, 3.*, March 1983, 253-263.

[Hart68] Hart, P., Nillsson, N.J., Raphael, B., A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Trans of Systems Science and Cybernetic, vol. ssc-4, No.2.*, July 1968.

[Henl88] Henley, A.J., Navigation and Guidance Using a Terrain Database, *A RAeS Symposium 'Guidance and Control Systems for Tactical Weapons'*, April 1988, 169-183.

[Henl92] Henley A.J., Milligan, J., Applications of Terrain and Feature Database to Aircraft Operations, *RIN & DGON Digital Mapping and Navigation conference*, London, Nov 1992, paper No.45.

[Herb91] Herbelin, R.L., Weber, J.W., Small, D.M., Airborne Electronic Terrain Map System - Conclusions, *IEEE National Aerospace and Electronics Conference* 1984, 1301-1307.

[Hewi91] Hewitt, C., Henley, A.J., Boyes, J.D., A Ground and Obstacle Collision Avoidance Technique (GOCAT), *IEEE National Aerdspace and Electronics Conference (NAECON)* 1991, 604-610.

[Hunt78] Hunter, G.M., *Efficient Computation and Data Structures for Graphics*, Ph.D Thesis, Princeton University, 1978.

[Hunt79] Hunter, G.M., Steiglitz, K., Operations on Images Using Quadtrees, *IEEE Transactions on Pattern Analysis, Machine Intelligence 1, 2.*, April 1979, 145-153.

[Kamb86] Kambhampati, S., Davis, L.S., Multiresolution Path Planning for Mobile Robots, *IEEE Journal of Robotics and Automation, vol. RA-2, No. 3.*, Sep 1986, 135-145.

[Kayt69] Kayton M., Fried W.R., *Avionics Navigation Systems*, John Wiley & Sons, New York, 1969.

[Keir84] Keirsey, D.M., Mitchell, J.S.B., Planning Strategic Paths Through Variable Terrain Data, *SPIE vol. 485, Application of Artificial Intelligence*, 1984.

[Keir84] Keirsey, D.M., Mitchell, J.S.B., Payton, D.W., Preyss, E.B., Multilevel Path planning for Autonomous Vehicles, *SPIE vol 485 Applications of Artificial Intelligence* 1984, 133-137.

[Klin76] Klinger, A., Dyer, C.R., Experiments in Picture Representation Using Regular Decomposition, *Computer Graphs and Image Processing 5, 1*, Jan 1976, 68-105.

[Klin79] Klinger, A., Rhodes, M.L., Organization and Access of Image Data by Areas, *IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.PAMI-1., No. 1.* Jan 1979, 50-60.

[Lato91] Latombe, J.C., *Robot Motion Planning*, Kluwer Academic Publishing, Boston, 1991.

[Lau87] Lau, W.K., Bernsteen, S.A., Fine, B.T., Integrated Terrain Access/Retrieval System (ITARS) Robust Demonstration System, *IEEE National Aerospace and Electronics Conference (NAECON)* 1987, 66-72.

[Latt91] Lattanzi, M.R., Shaffer, C.A., NOTE, An Optimal Boundary to Quadtree Conversion Algorithm, *Computer Vision, Graphics, and Image processing 53, 3.*, May 1991, 303-312.

[Lauz85] Lauzon, J.P., Mark, D.M., Kikuchi, L., Guevara, J.A., Two-dimensional Run-Encoding for Quadtree Representation, *Computer Graphics and Image Processing 30, 1.*, April 1985, 56-69.

[Leig84] Leightly, R.D., Terrain Navigation Concepts for Autonomous Vehicles, *SPIE vol 485 Applications of Artificial Intelligence* 1984, 120-125.

[Levi80] Levine, M.D., Region Analysis Using a Pyramid Data Structure - *Structure Computer Vision*, Academic Press, Inc., 1980, 57-100.

[Lizz85] Lizza, C.S., Lizza, G., Path-Finder: An Heuristic Approach to Aircraft Routing, *IEEE National Aerospace and Electronics Conference* 1985, 1436-1443.

[Loza79] Lozano-Perez, T., Wesley, M., An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles, *Communications of the ACM 22,10.*, Oct 1979, 560-570.

[Loza81] Lozano-Perez, T., Automatic Planning of Manipulator transfer movements, *IEEE Transactions on Systems, Man and cybernetics, vol.11*, 1981, 681-698.

[Loza83] Lozano-Perez, T., Spatial Planning : A configuration Space Approach, *IEEE Transactions on Computers, C32(2)*, 1983, 108-120.

[Lux90] Lux, P., Eibert, M., ISS - A Combined Terrain Topograph Referenced Navigation System, *IEEE Position, Location and Navigation Symposium* 1990, 470-473.

[Mark84] Mark, D.M., Lauzon, J.P., Linear Quadtrees for Geographic Information Systems,. *Proceedings, International Symposium. Spatial Data Handling*, Zurich, Switzerland, August 1984, 412-430.

[Mark85a] Mark, D.M., Lauzon, J.P., The Space Efficiency of Quadtree: An empirical Examination Including the Effects of Two-Dimensional Run-

Encoding, *Geo-Processing 2,* 1985, 367-383.

[Mark85b] Mark, D.M., Abel, D., Linear Quadtrees from Vector Representations of Polygons, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7,3.*, May 1985, 344-349.

[Mark86] Mark, D.M., The Use of Quadtrees in Geographic Information Systems and Spatial Data Handling, *Proceedings of Auto-Carto London, vol. 1*, London, Sep 1986, 517-526.

[Meng87] Meng, A.C-C., Flight Path Planning Under Uncertainty for Robotic Air Vehicles, *IEEE National Aerospace and Electronics Conference (NAECON)* 1987, 359-366.

[Meno87] Menon S., Gao, P., Smith, T.R., Multi-coloured Quadtrees for GIS: Exploiting Bit-Parallelism for Rapid Boolean Overlay, *Proceedings of the International Symposium on GIS, vol. 2,* Arlington, VA, November 1987, 371-383.

[Midd89] Middleton, D.H., *Avionic Systems*, Longman Scientific & Technical, Essex, 1989.

[Mitc84a] Mitchell, J.S.B., An autonomous Vehicle Navigation Algorithm, *SPIE vol. 485 Applications of Artificial Intelligence* 1984, 153-158.

[Mitc84b] Mitchell, J.S.B. *Planning Shortest Paths,* Ph.D Thesis, Stanford University, Aug 1986.

[Mitc88] Mitchell, J.S.B., An Algorithmic Approach to Some Problems in Terrain Navigation, *Artificial Intelligence, Vol 37 no 1-3*, 1988, 171-201.

[Mort66] Morton, G.M. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequence*, IBM Ltd., Ottawa, Canada, 1966.

[Newm78] Newman,W.M., Sproull, R.F., *Principles of Interactive Computer Graphics*, McGraw-Hill, Inc., 1978.

[Nils82] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga, Palo Alto., 1980.

[O'Dun82] O'Dunlaing, C., Yap, C.K., A Retraction Method for Planning the Motion of a Disc, *Journal of Algorithms, 6,* 1982, 104-111.

[Oska90] Oskard, D.N., Hong, T.H., Shaffer, C., Real-time Algorithms and Data Structures for Underwater Mapping, *IEEE Transactions on Systems, Man, and Cybernetics, vol. 20, No. 6.*, Nov/Dec 1990, 1469-1475.

[Payl93] Paylor, A., *Air Traffic Control, Today & Tomorrow*, Ian Allan Publishing, 1993.

[Pau90] Pau, L.F., Mapping and Spatial Modelling for Navigation, *Proceedings of the NATO Advanced Research Workshop*, Springer-Verlag, Berlin, 1990.

[Prie90] Priestley, N., Terrain Reference Navigation, *IEEE Position, Location and Navigation Symposium* 1990, 482-489.

[Rich86] Rich, A., *Artificial Intelligence*, McGraw-Hill, Inc., 5th ed, 1986.

[Rodd90] Roddriguez, J.J., Aggarwal, J.K., Matching Aerial Images to 3-D Terrain Maps, *IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, No.12.*, Dec 1990, 1138-1149.

[Salz88] Salzberg, B., *File Structures, An Analytic Approach*, Prentice-Hall 1988.

[Same80a] Samet, H., Region Representation: Quadtrees from Binary Arrays, *Computer Graphics and Image Processing 13, 1.*, May 1980, 88-93.

[Same80b] Samet, H., Region Representation: Quadtrees from Boundary Codes, *Communication of the ACM 23, 3.*, March 1980, 163-170.

[Same81] Samet, H., An Algorithm for Converting Rasters to Quaftrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 3, 1.*, Jan 1981, 93-95.

[Same82] Samet, H., Neighbour Finding Techniques for Images Represented by Quadtrees, *Computer Graphic and Image Processing 18,1.*, January 1982, 37-57.

[Same84a] Samet, H., Rosenfeld, A., Shaffer, C.A., Use of Hierarchical Data Structures in Geographical Information Systems, *IEEE Transactions PAMI-7, No. 3.*, May 1984, 392-403.

[Same84b] Samet, H., Webber, R.E., On Encoding Boundaries with Quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6, 3.*, May 1984, 365-369.

[Same84c] Samet, H., Rosenfeld, A., Shaffer, C. A., and Webber, R. E., A *Geographic Information System Using Quadtrees, Pattern Recognition 17, 6.*, November 1984, 647-656.

[Same85] Samet, H., Tamminen, M, Computing Geometric Properties of Images represented by Linear Quadtrees, *IEEE Transactions on Pattern Analysis*

and Machine Intelligence, 7, 2., March 1985, 229-240.

[Same89] Samet, H., Neighbour Finding Techniques for Images Represented by Octrees, *Computer Vision, Graphic and Image Processing 46,3.*, June 1989, 367-386.

[Same90a] Samet, H., *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[Same90b] Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[Scha81] Schachter, B.J., Computer Image Generation for Flight Simulation, *IEEE CG&A*, October 1981, 29-68.

[Schw88] Schwartz, J.T., Sharir, M., A survey of Motion Planning and Related Geometric Algorithms, *Artificial Intelligence 37*, 1988, 157-169.

[Shaf89] Shaffer, C.A., Samet, H., Nelson, R.C., *Technique Report, QUILT: A Geographic Information System Based on Quadtrees*, Feb 1989, University of Maryland.

[Shar86] Sharir, M., Schorr, A., On Shortest Paths in Polyhedral Spaces, *SIAM Journal of Computer. 15 (1)* 1986, 193-215.

[Shne81] Shneier M., Calculations of Geometric Properties Using Quadtrees, *Computer Graphics and Image Processing 16.*, 1981, 296-302.

[Siou93] Siouris, G.M., *Aerospace Avionics Systems, A modern Synthesis*, Academic Press, Inc., 1993.

[Silb91] Silbert M., Comparison of the Even-Step Algorithm to Other Path Planning Methods to Avoid Dynamic 3D Obstacles, *HGARD, GCP 53*, Oct 1991.

[Sloa79] Sloan, K.R., Tanimoto, S.L., Progressive Refinement of Raster Images, *IEEE Transactions on Computers 28, 11.*, November 1979, 871-874.

[Tani75] Tanimoto, S.L., Pavlidis, T., A Hierarchical Data Structure for Picture *Processing, Computer Graphics and Image Processing, vol. 4, No. 2.*, Jan 1975, 104-119.

[Tani76] Tanimoto, S.L., Pictorial Feature Distortion in A Pyramid, *Computer Graphics and Image Processing vol. 5, No. 3*, 1976, 333-352.

[Tani80] Tanimoto, S.L., *Image Data Structure - Structure Computer Vision,*

Academic Press, Inc., 1980, 31-55.

[Udup77] Udupa, S., *Collision Detection and Avoidance in Computer Controlled Manipulators*, Ph.D Thesis, California Institute of Technology, 1977.

[Waru84] Waruszewski, H., Defense Mapping Agency Digital Data Bases, *IEEE National Aerospace and Electronics Conference* 1984, 70-75.

[Webe84] Weber, J.W., Future Electronic Map Systems, *IEEE National Aerospace and Electronics Conference* 1984, 106-111.

[Whit69] White, D.J., *Dynamic Programming*, Oliver & Boyd LTD, Edinburgh, 1969.

[Wong86] Wong, E.K., Fu, K.S., A Hierarchical Orthogonal Space Approach to Three Dimensional Path Planning, *IEEE Journal of Robotics and Automation, vol. RA-2, No. 1.*, March 1986, 42-53.

[Zele92] Zelenka, R.E., Integration of Radar Altimeter, Precision Navigation, and Digital Terrain Data for Low-Altitude Flight, *AIAA-92-4420-CP*, 1992, 605-615.

APPENDIX 1 A* Algorithm

A variant of the A* algorithm is also implemented with the evaluation function f of a node N defined as $f(N) = g(N) + h(N)$ to find a shortest flight path, where $g(N)$ represents the distance of the path segments from the start point to N, $h(N)$ represents the heuristic estimate of the distance of the remaining path from N to the goal point which is calculated as the Euclidean distance between N and the goal node.

The input consists of a list of path segemnts L, N_{start}, N_{goal} and h. The list L is obtained from the function SETUP(W_{points}) as described in section 4.5.2. Each record in the list L contains the information of W_{from} and W_{to} waypoints of a path segment, the distance of a path segment and a flag used in backtracking. All the flags are initially marked *unvisited*. The algorithm makes use of a list denoted by OPEN that contains path segments sorted by the values of the function f. The list OPEN supports the following operations:

- FIRST(OPEN): remove the node of OPEN with the smallest value of f and return it,
- INSERT(N,OPEN): insert node N in OPEN,
- DELETE(N,OPEN): remove node N from OPEN,
- MEMBER(N,OPEN): node N is in OPEN return TRUE otherwise return FALSE,
- EMPTY(OPEN): if OPEN is empty return TRUE otherwise return FALSE.

The A* algorithm explores a graph iteratively by following paths originating at N_{start}. At the beginning of every iteration, there are some nodes that the algorithm has already visited, and there may be others that are still unvisited. For each visited node N, the previous iterations have produced one or more paths connecting N_{start} to N, but the algorithm only memorizes a representation of a path of minimum cost among those so far constructed. At any instant, the set of all such paths forms a spanning tree T of the subset of graph so far explored. T is represented by associating with each visited node N a pointer to its parent node in the current T. At each iteration, A* examines the nodes adjacent to the node returned by function FIRST(OPEN). Initially, both the spanning tree T and the list OPEN are empty. The A* algorithm is given below.

```
procedure A*(L,N_start,N_goal);
int N,N', distance;
begin
    assign N_start into T;
    INSERT(N_start,OPEN); mark N_start visited;
    while EMPTY(OPEN) do
    begin
      N ← FIRST(OPEN);
      if N = N_goal then exit while-loop;
      for every path segment (N,N') in L do
        if (N,N') is not visited then
          begin
            add N' to T with a pointer toward N;
            INSERT(N',OPEN); mark (N,N') visited;
          end;
        else if g(N') > g(N) + distance(N,N') then
          begin
            modify T by redirecting the pointer of N' toward N;
            if MEMBER(N',OPEN) then DELETE(N',OPEN);
              INSERT(N',OPEN);
          end;
end;
  if EMPTY(OPEN) then
    return the constructed path by tracing the pointers in T from N_goal back to N_start;
  else return failure;
end;
```

The examination of the nodes adjacent to a node N in the main loop of the algorithm is called the expansion of N. When the expansion of N produces a 'visited' node N'. The path including N' may provide a path that is less costly than any of the previously generated paths from N_{start} to N'. Then, the algorithm updates T by redirecting the pointer issued from N'. If N' is in OPEN, its position in the list must be updated according to the new value of $f(N')$. If N' is not in OPEN, it may happen that the better path discovered between N_{init} and N' also provides better paths for attaining previously visited successors of N' which are currently not successors of N' in T; rather than modifying the path immediately, the algorithm reinserts N' in OPEN, so that its successors will be reconsidered later.